Advanced Neural Network Engine Control

Luther, Jim Benjamin

Publication date:
2005

Document Version
Early version, also known as pre-print

Advanced Neural Network Engine Control

Jim Benjamin Luther

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
TECHNICAL UNIVERSITY OF DENMARK
LYNGBY, DENMARK
November 2002

ØRSTED·DTU, Automation
DTU, Bld. 326
DK-2800 Kgs. Lyngby
Denmark
Phone +45 4525 3550

Printed in Denmark at DTU, Kgs. Lyngby
01-A-918
ISBN 87-87950-90-1

TECHNICAL UNIVERSITY OF DENMARK
DEPARTMENT OF AUTOMATION

The following persons, signing here, hereby certify that they have read and recommend to the Konsistorium the acceptance of a dissertation entitled „**Advanced Neural Network Engine Control**“ by **Jim Benjamin Luther** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: November 2002

External Examiner: _____

Research Supervisors: _____
Elbert L. Hendricks

_____

Examining Committee: _____

TECHNICAL UNIVERSITY OF DENMARK

November, 2002

Author:	**Jim Benjamin Luther**
Title:	**Advanced Neural Network Engine Control**
Department:	**Department of Automation**
Degree:	**Ph.D.** Convocation: **-** Year: **2002**

Signature of Author

For the love of my life... The sunshine in my life...

Wahyuning Diah

For my parents, my family and my friends

List of Figures

1.1 Idealized neural activation function 5
1.2 Single Hidden Layer Neural Network 6

2.1 Air Flow Process 19
2.2 F_{in} Controller 24
2.3 Operating Points For Exhaust Tests 25
2.4 No EGR Training and Test Results - With Exhaust Temperature . 29
2.5 No EGR Training and Test Results - Without Exhaust Temperature 30
2.6 EGR Training and Test Results - With Exhaust Temperature . . . 31
2.7 EGR Training and Test Results - Without Exhaust Temperature . . 32
2.8 NGK Oxygen Sensor Characteristics 34
2.9 Absolute Training and Test Set Errors 37
2.10 Test Set Relative Error 38
2.11 Operating Points 40
2.12 Peak Pressure Results - One Output 43
2.13 Peak Pressure Location - One Output 44
2.14 Peak Pressure Results - Two Outputs 45
2.15 Peak Pressure Location Results - Two Outputs 46

3.1 Dynamic Neural Network Setup 52
3.2 One Step Ahead Training Results 70
3.3 One Step Ahead Training - Simulation Example 70
3.4 Five Step Ahead Predictive Training - Simulation Example 71

4.1 The throttle air mass flow and its MVEM mean value signal 90
4.2 Si as a Function of n for the ECG Test Engine 94
4.3 Yi as a Function of n for the ECG Test Engine 95
4.4 A section from a data set where the intake manifold pressure exceeds the ambient pressure 97
4.5 Mirrored and Scaled β_2 Function 98
4.6 A Continuous tanh() Extension of the β_2 Function 101

4.7 AMVEM Outputs Compared to a Measured SI Engines Signals - No EGR . . . 104
4.8 AMVEM Outputs Compared to a Measured SI Engines Signals - EGR . . . 105
4.9 AMVEM Simulation and Measured Data . . . 107
4.10 AMVEM Air Mass Flows and Measured Data . . . 108
4.11 Throttle Plate Angle - Speed Steps . . . 112
4.12 Throttle Plate Angle - Throttle Plate Angle Steps . . . 113
4.13 Reference Speed Step Patterns . . . 114
4.14 Fall Out Removal . . . 116
4.15 Throttle Air Mass Flow Simulations . . . 120
4.16 Throttle Air Mass Flow Simulations . . . 121
4.17 A Close Up on the Manifold Filling and Emptying Spikes for the Test Data Set Simulation . . . 122
4.18 Lambda Sensor Error . . . 123
4.19 Lambda Simulations - Training and Test Sets . . . 124
4.20 Intake Manifold Pressure Simulations - Better Than AMVEM . . . 127
4.21 Intake Manifold Pressure Simulation - Worse . . . 128
4.22 Intake Manifold Temperature Simulations - Better Than AMVEM 130
4.23 Intake Manifold Temperature Simulation - Test Set . . . 131

5.1 Predictive Control Block Diagram . . . 134
5.2 MIMO NPC Test Model . . . 162
5.3 MIMO NPC Test Model Input Signals . . . 163
5.4 MIMO NPC Test Model and Neural Network Output Signals . . . 164
5.5 MIMO NPC Control Demonstration Set Up . . . 164
5.6 MIMO NPC Control Demonstration on the Test Model . . . 165
5.7 H_2 Controller Set Up . . . 167
5.8 MIMO NPC and H_2 Controller Comparison . . . 170

6.1 A simulation using the cost function without a final state cost added. 179
6.2 A simulation using the cost function with the final state cost added. 180

7.1 Variance Dependent Filter Demonstration . . . 189
7.2 A Two Layer Neural Network . . . 191

A.1 Neural Network Toolbox Graphical User Interface . . . 194
A.2 The Extraction GUI . . . 201
A.3 PCON Data Converter Program . . . 202

List of Tables

2.1 Variables Measured. 22
2.2 Exhaust O_2 Feature Table 26
2.3 In-Cylinder Neural Network Estimation Inputs 33
2.4 Operating Points. 40

3.1 Throttle Air Mass Flow Example Neural Network Configuration . 69

4.1 British Leyland 1.275 L Engine Specifications 91
4.2 Input Output Signal Name List 118

5.1 Test System Neural Network Model Training Parameters 163
5.2 MIMO NPC Control Demo Parameters 165
5.3 H_2 Controller Parameters 169

A.1 Neural Pressure Data File Matrix Format. 199
A.2 Matrix Constructors. 204
A.3 Matrix Operators. 206
A.4 Matrix External Operators. 207
A.5 Matrix Member Functions. 207
A.6 Row and Column Operations. 208
A.7 Matrix Utility Functions. 209
A.8 Matrix Information Functions. 210
A.9 Mathematical Matrix Functions. 210
A.10 Marquardt Class Optimization Parameters 214
A.11 Marquardt Member Functions. 216
A.12 Marquardt Default Parameters 217
A.13 Input-output set example. 220
A.14 SHLNetwork Member Functions 226
A.15 SHLNetwork Default Parameters 227
A.16 NetworkTrainer Member Functions 228
A.17 SHLNetwork Default Parameters 229
A.18 PredictiveNetworkTrainer Member Functions 232

A.19 PredictiveController Constructor Parameters 236
A.20 PCB Members Variables 237
A.21 Predictive Controller Member Functions 238
A.22 Neural Predictive Controller Member Functions 239
A.23 NPC MEX Function Parameters 240

Abstract

The main subject of this dissertation is the dynamic modelling of Spark Ignition (SI) engines. This is done on the level of Mean Value Engine Models (MVEMs). This modelling is done using physical modelling techniques as well as using neural networks. One of the main aims of the dissertation is to compare these two modelling techniques with respect to accuracy. Engine subsystems are modelled by dynamic neural networks trained with the predictive neural network training algorithm developed in this work and the results are compared with the physical Adiabatic Mean Value Engine model's corresponding signals.

In addition, neural networks' accuracy as virtual sensors for O_2 concentration in the exhaust, in-cylinder air fuel ratio, in-cylinder peak pressure and peak pressure location is tested for a 2.0 L Puma diesel engine. A predictive cost function based neural network training algorithm for improved dynamic neural network model training has been developed. Furthermore, the necessary mathematics for a fast C++ implementation as well as a C++ implementation of a matrix library and the predictive neural network training algorithm has been developed.

A general neural network based multi input multi output predictive controller algorithm has been developed and the necessary mathematics for a fast C++ implementation is derived. A C++ implementation of the neural network based predictive controller has been developed. A proof of stability is given for the predictive control strategy utilized if a final state cost is added to the cost function.

Dansk resumé

Denne afhandlings hovedemne er dynamisk modellering af benzin motorer. Dette er baseret på middelværdi motor model (MVEM) strukturen. Modelleringen er baseret på fysiske modelleringsteknikker og neurale netværk. En af de vigtigste emner i afhandlingen er en sammenligning af disse to modelleringsmetoder med hensyn til nøjagtigheden af modellerne. Motorens delsystemer er blevet modelleret af dynamiske neurale netværk, som er blevet trænet med en prædiktiv neural netværk trænings algoritme udviklet og beskrevet i dette arbejde. Resultaterne bliver sammenlignet med de tilsvarende middelværdi motor modellers signaler.

Yderligere er neurale netværks nøjagtighed som virtuelle sensorer for O_2 koncentrationen i udstødningen, luft brændstof forholdet i cylinderne, største tryk og største tryks position i cylinderne for en 2.0 L Puma diesel motor undersøgt. En prædiktiv kostfunktion baseret neural netværks trænings algoritme er blevet udviklet for at forbedre den dynamiske neural netværks træning. Der er også blevet udviklet den nødvendige matematik til at implementerer et hurtigt C++ prædiktivt neuralt netværks trænings program samt en C++ udgave af et matrix program bibliotek og den prædiktive neurale trænings algoritme.

En general neural netværks baseret multi input multi output prædiktiv regulator algoritme og den nødvendige matematik til en hurtig C++ version er blevet udviklet. En C++ udgave af den neurale netværks baserede regulator er blevet udviklet. Et bevis for stabilitet af den prædiktive regulator strategi er ført hvis et slut tilstandsled bidrag bliver tilføjet til kostfunktionen.

Acknowledgements

No words are accurate enough to express what my Wahyuning Diah means to me... She creates light in my heart expelling all sadness when everything sometimes seems too much...Terima Kasih Banyak Sayangku...

My sincere thanks goes to Christian Winge Vigild for always calling me up to hear how I am even though I was too tired to remember to call back. I am also very grateful for all the technical input that Christian always gave me during those calls. It made me feel less alone with my work...

I am very grateful for all the help that Rolf Poder and Marco Fam has given me with the test engine and many other things.

And thank you so much Elbert Hendricks for keeping me going. I could not have done that without your help...

Jim Benjamin Luther

Copenhagen, Denmark
November, 2002

Contents

List of Figures x

List of Tables xii

Abstract xiv

Dansk resumé xvi

Acknowledgements xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Organization and Contributions . . . 3
- 1.3 A Brief Introduction to Neural Networks . . . 4
  - 1.3.1 Complexity and Accuracy . . . 4
  - 1.3.2 Network Training . . . 7
  - 1.3.3 Modelling . . . 8
  - Simplicity and preservation . . . 9
- 1.4 Related Work . . . 10
  - 1.4.1 Virtual Sensors . . . 10
  - 1.4.2 Neural Network Model Based Controllers . . . 11
  - Spark advance control . . . 11
  - Mixed Controllers . . . 12
  - 1.4.3 Direct Neural Control . . . 13
  - 1.4.4 Predictive Neural Network Model Based Control . . . 14

2 Virtual Sensors 17
- 2.1 Introduction . . . 17
- 2.2 Data Gathering . . . 17
  - 2.2.1 Training Data . . . 18
  - 2.2.2 Precautions . . . 18

2.2.3 In-Cylinder Lambda Measurement . . . 18
Estimating the Lambda Value in the Cylinder . . . 18
2.2.4 Data Points and Signals . . . 21
Operating Points . . . 21
Measured Variables . . . 22
2.2.5 Fin Controller . . . 23
2.3 Estimation of O2 in the Exhaust . . . 25
2.3.1 Purpose . . . 25
2.3.2 Setup . . . 25
Network inputs . . . 26
Training . . . 27
2.3.3 Results and Discussion . . . 27
No EGR training, no EGR testing - Utilizing The Exhaust Temperature . . . 29
No EGR training, no EGR testing - Not Utilizing The Exhaust Temperature . . . 30
EGR training, no EGR testing - Utilizing The Exhaust Temperature . . . 31
EGR training, no EGR testing - Not Utilizing The Exhaust Temperature . . . 32
2.4 Estimation of Air Fuel Ratio in the Cylinder . . . 33
2.4.1 Purpose . . . 33
2.4.2 Setup . . . 33
2.4.3 Results . . . 35
2.4.4 Discussion . . . 35
In Cylinder Lambda Estimation - Absolute Errors . . . 37
In Cylinder Lambda Estimation - Test Set Relative Errors . 38
2.5 Estimation of Peak and Peak Pressure Location . . . 39
2.5.1 Purpose . . . 39
2.5.2 Setup . . . 39
Creating data sets . . . 39
2.5.3 Results . . . 40
2.5.4 Discussion . . . 41
Peak Pressure Results - Single Output Networks . . . 43
Peak Pressure Location Results - Single Output Networks 44
Peak Pressure Results - Two Output Network . . . 45
Peak Pressure Location Results - Two Output Network . . 46
2.6 Conclusions . . . 47
2.6.1 Exhaust O2 Concentration . . . 47
2.6.2 In-Cylinder Air Fuel Ratio . . . 47
2.6.3 Peak Pressure and Peak Pressure Location . . . 48

2.6.4 Overall Virtual Sensor Conclusions 48

3 Dynamic Neural Network Model Training **51**
3.1 Basic Dynamic Neural Network Model Training 51
3.1.1 Dynamic Neural Networks 51
3.1.2 Dynamic Neural Network Training 53
A General Dynamic Neural Network Function 55
Neural Network Model Training Problems 56
The Naive Predictor Case (1) 57
The Neural Network Model Becomes Unstable (2) 57
The Neural Network Model Output Diverges in Some Areas (3) 58
3.2 Predictive Neural Network Model Training 58
3.3 Predictive Neural Network Training Algorithm 59
3.3.1 The Predictive Cost Function 60
3.3.2 The predictions - A Closer Look - Derivatives 62
3.3.3 Dynamic Neural Network Partial Derivatives 64
Sparse Matrix Product 66
3.3.4 Demonstration of the Predictive Training Algorithm . . . 68
3.4 Levenberg Marquardt Algorithms Literature 69
3.4.1 Background 69
3.4.2 Comparison of LM algorithms in the Literature with the Predictive Training Algorithm 72
Other Implementations of the LM Algorithm in the Literature 73
The Utilization of the LM Algorithm in the Predictive Training Algorithm 73
3.5 Neural Network Derivatives 74
3.5.1 The Neural Network Output Function 74
3.5.2 The Derivative with Respect to the Weights 76
3.5.3 The Derivative with Respect to the Inputs 77
3.5.4 Multiple Input Sets 78
3.5.5 Neural Network Derivative Matrix Formats 79
3.5.6 Programming the Derivatives 80
3.5.7 Programming the Derivative with Respect to the Weights . 81
3.5.8 Programming the Derivative with Respect to the Inputs . . 85
3.6 Conclusions 87
3.6.1 Predictive Training Algorithm 87

4 Mean Value Engine Modelling (MVEM) 89
4.1 MVEM Introduction . . . 89
4.2 The ECG Test Engine . . . 90
4.3 Common MVEM equations . . . 91
4.3.1 The Crank Shaft Speed . . . 91
4.3.2 The Port Air Mass Flow . . . 93
4.3.3 The Throttle Air Mass Flow . . . 94
Continuous BETA2 Extension . . . 96
Continuous β_2 Soft Extension . . . 98
4.3.4 BETA2 Extensions Comments . . . 100
4.4 The Intake Manifold Equations . . . 100
4.4.1 Isothermal Equation . . . 101
4.4.2 Adiabatic Equations . . . 102
4.5 The AMVEM Accuracy in the Literature . . . 102
4.5.1 Intake Manifold Modelling Accuracy . . . 103
4.5.2 Lambda Modelling Accuracy . . . 106
4.6 The AMVEM Accuracy in a Wider Operating Area . . . 106
4.7 Neural MVEM modelling . . . 108
4.7.1 Introduction . . . 108
4.7.2 Training and Test Data Set Generation . . . 110
4.7.3 Fallout Removal . . . 115
4.7.4 Temperature Sensor Problems . . . 117
4.8 Engine Subsystems Neural Network Modelling . . . 117
4.8.1 Input Output Signals Symbol List . . . 118
4.8.2 Throttle Air Mass Flow Modelling . . . 118
4.8.3 Lambda Modelling . . . 122
4.8.4 Cranks Shaft Speed and Lambda MIMO Modelling . . . 125
4.8.5 Intake Manifold Pressure Modelling . . . 125
4.8.6 Intake Manifold Temperature Modelling . . . 128
4.9 Conclusions . . . 131

5 MIMO Neural Predictive Control 133
5.1 MIMO Nonlinear Predictive Controller . . . 133
5.1.1 Algorithm Overview . . . 133
Predictive control block diagram . . . 133
Block diagram description . . . 135
5.1.2 A Nonlinear MIMO Model . . . 135
The Model . . . 136
Shorthand . . . 137
Shorthand Example . . . 137
5.1.3 The Cost Function . . . 138

Predictive Cost Function . . . 138
Vector Formats . . . 139
5.1.4 Optimization Algorithm . . . 141
Multiple Quadratic Term Cost Function. . . . 141
Second Order Approximations . . . 142
Minimization Step . . . 143
Note on Multiple Lambdas . . . 145
5.1.5 Prediction Derivatives . . . 145
Iterative Prediction . . . 146
Future Control Signal Time Limit . . . 146
Efficiency . . . 148
Splitting Up the Inputs of the Model Function . . . 148
Predictor Input Structure . . . 149
Calculating the Individual Derivatives . . . 150
Iterative Prediction Derivative Calculation . . . 152
Constant Future Controls . . . 153
Iterative Derivative Algorithm - Overview . . . 154
The Future Control Vector Format . . . 156
The Control Cost Derivative . . . 157
5.1.6 System Model Derivatives . . . 157
5.1.7 Pseudo Code For the Controller . . . 158
5.2 Demonstration . . . 161
5.2.1 The Test System . . . 161
5.2.2 Test Model Neural Network Training . . . 161
Training Data Generation . . . 161
Neural Network Training . . . 162
5.2.3 MIMO Nonlinear Predictive Controller Demonstration . . . 163
5.2.4 Comparison With an H2 Controller . . . 166
Linearized State Space Form . . . 166
H2 Controller Setup . . . 167
H2 and MIMO NPC Controller Comparison . . . 169
5.2.5 Conclusions . . . 170

6 Neural Network Based Controller Stability **173**
6.1 Introduction . . . 173
6.2 The Proof . . . 174
6.2.1 The Cost Function . . . 175
6.2.2 Assumptions . . . 176
6.2.3 Rewriting the Cost Function . . . 177
6.2.4 A Neural Model . . . 178
6.3 Example . . . 178

6.4 Conclusion 179

7 Overall Conclusions **181**
7.1 Conclusion 181
7.1.1 Neural Network Virtual Sensors 181
Exhaust O2 Concentration 181
In-Cylinder Air Fuel Ratio 182
Peak Pressure and Peak Pressure Location 182
Overall Virtual Sensor Conclusions 183
Software 183
7.1.2 Dynamic Neural Network Training 183
Predictive Training Algorithm 183
Software 184
7.1.3 Neural Network Engine Modelling 184
7.1.4 Neural Predictive Controller 186
7.1.5 Stability of Predictive Control Strategy 186
7.2 Suggestions for the Future 186
7.2.1 Sensors 186
7.2.2 Data Sets 187
7.2.3 Data Filtering 187
Variance Dependent Filter 188
7.2.4 Predictive Training Algorithm 189
7.2.5 Neural Network Structure 190
7.2.6 MIMO Nonlinear Predictive Control Algorithm 191
7.2.7 PredictiveController Class 192

A Tools Used and Developed **193**
A.1 IAU Neural Network Toolbox 193
A.2 Neural network graphical user interface 193
A.2.1 How to use it 194
Selecting data files 194
Setting network parameters 195
Setting the training parameters 196
Training the network 197
Network parameter utilities 198
A.2.2 Train and test file format 199
A.3 The extraction program 200
A.4 Data Conversion Programs 201
A.4.1 INDISET data conversion 201
A.4.2 Further development issues 202
A.4.3 PUMA data file conversion 203

A.5 C++ Matrix Library . . . 203
A.5.1 Purpose . . . 203
A.5.2 Initialization . . . 203
A.5.3 Matrix operators . . . 204
Binary and unary operators . . . 204
External operators . . . 204
A.5.4 Matrix functions . . . 204
Information functions in the Matrix class . . . 204
Information functions outside the class . . . 205
A.5.5 Examples . . . 211
Initialization . . . 211
Matrix arithmetics . . . 212
A.5.6 No checking of any kind in the library! . . . 212
A.6 Optimization Library . . . 213
A.6.1 Purpose . . . 213
A.6.2 How to Use the Library . . . 213
A.6.3 Marquardt Member Functions . . . 215
A.6.4 Derivative Functions. . . . 217
A.6.5 Example . . . 218
A.7 Neural Network Library . . . 219
A.7.1 The Single Hidden Layer Network Class . . . 220
Input-Output Sets . . . 220
Input Matrix Format . . . 220
Output Matrix Format . . . 221
Weight Matrices Format . . . 221
Weight Vector Data Format . . . 223
Order of the Derivative Outputs Elements . . . 224
SHLNetwork member functions . . . 225
SHLNetwork Default Values . . . 227
A.7.2 The Network Trainer Class . . . 227
NetworkTrainer Member Functions . . . 227
The Training Input and Output Data Format . . . 229
A.7.3 NetworkTrainer Default Parameters . . . 229
A.7.4 Example . . . 229
A.7.5 The Predictive Network Trainer Class . . . 231
A.8 Control Library . . . 233
A.8.1 Introduction . . . 233
A.8.2 A Brief Description of Predictive Control Strategy . . . 233
A.8.3 PredictiveController Class Description . . . 235
A.8.4 NeuralPredictiveController Class Description . . . 239
A.8.5 Neural Predictive Controller MATLAB MEX Function . . . 239

A.9 The Automatic Fallout Removal MATLAB Program 241

Chapter 1

Introduction

1.1 Motivation

The demands for very accurate engine control are currently increasing rapidly. Governments everywhere are concerned with the amount of pollution that is being emitted into the biosphere. With good reasons. Doctors report of an increased number of asthma and other lung related diseases. Geologists report of an increased average temperature on earth and that it is still increasing. Just recently, an area the size of the Danish isle Bornholm, broke loose from the Antarctic continent and the size of the ice cap covering the south pole has been reduced significantly. The ice is melting. This is thought by many scientist to be caused by global warming as a result of too much CO_2 being emitted into the atmosphere.

It is clear from these observations and because of the large number of cars in the world today that it is important to reduce the amount of pollution coming from the intensive use of internal combustion engines and diesel engines.

The pollution coming from those engines can be split into two categories.

1. Toxic gasses and particles which cause lung diseases and other kinds of diseases.

2. CO_2 which heats up the planet.

Nowadays for gasoline engine based cars it is becoming mandatory in more and more countries to have a catalyst in cars. The catalysts' job is to reduce the amounts of toxic gasses in the exhaust gas. The catalysts, however, can not do this properly unless the the gasoline has been burned at the stoichiometric ratio for air and fuel. It is therefore very important that the fuel injection is controlled

very accurately so that this ratio is always maintained. How well this can be done depends how accurately the engine can be modelled and the capabilities of the controller.

The same holds for diesel engine based cars. In this case, however there is also another problem. If the diesel engine does not get enough air or burns the fuel at too low a temperature, then the fuel will not burn properly and the combustion will thus produce a large amount of larger particles that are very unhealthy to breathe. It is therefore also very important in this case to know exactly what goes on in the diesel engine and to be able to provide the necessary control.

The amount of CO_2 being emitted can be reduced by using as little fuel as possible. To do that, one has also to have a very accurate model of the engine so that the exact amount of fuel necessary for the desired acceleration can be calculated. This pollution factor does of course also depend strongly on the hardware used to build the engine and the car itself. Engine control is however also an important factor in keeping the amount of fuel used down.

Up until now the engine controls most widely used for production cars has been mapping based controllers where the content of the maps is determined by trial and error rather than physical modelling or mathematical modelling systems such as neural networks. There are only a few systems based on linear models and simple controllers. This is both because of cost considerations and because of lack of trust in the usefulness of more advanced control concepts. It is the intention of this work to examine the advantages gained by using advanced nonlinear controllers and models for engine control and at the same time gain more insight into some important physical relationships in the engine itself which can also be used in simpler engine models.

Many nonlinear models and controllers exists. They all vary in complexity and capabilities. In this work the neural network based models and controllers have been chosen because of neural networks ability to uniformly approximate any continuous function with any desired accuracy. The networks will therefore provide ultimate freedom for both modelling and controlling purposes and are thus an interesting choice for this work.

1.2 Organization and Contributions

Chapter 1 contains a introduction to the dissertation, a description of how neural networks have been used in engine control recently and an introduction to neural networks. The chapter also briefly explains some interesting neural network applications of which one is developed for Multi Input Multi Output (MIMO) neural network based systems in chapter 4.

Chapter 2 is an analysis of neural networks ability as virtual sensors for engine diagnostics purposes. First a section about how the data for the virtual sensor research has been gathered and then three sections containing the analysis of a single hidden layer neural networks ability to estimate peak pressure and peak pressure location (In order to reduce the in-cylinder pressure sampling speed otherwise necessary), O_2 contents in the exhaust gas (in order to avoid the costly lambda sensor) and in-cylinder O_2 contents (Because no lambda sensor would work there and to achieve better combustion control).

Chapter 3 contains a description of how a dynamic neural network model is trained and the problems associated with that process. A new predictive neural network training algorithm for avoiding the the various training problems that can occur when training dynamical neural network models is suggested and the necessary mathematics for the development of a fast training algorithm is derived.

Chapter 4 is a description of a Mean Value Engine Model (MVEM) for a spark ignition (SI) engine and its accuracy with respect to real measured data. An adiabatic MVEM (AMVEM) is compared with neural network models of SI engine subsystems developed in this work. The neural network models are trained with the predictive training algorithm developed in chapter 3.

Chapter 5 describes in detail how the MIMO neural predictive controller algorithm has been developed and how it works. The controller is an extension of the one developed in [31] that was build for Single Input Single Output (SISO) systems. The MIMO capabilities is achieved by packing the multi dimensional data in a specific way compatible with the neural network derivatives with respect to inputs and weights developed in chapter 3.

Chapter 6 is about the stability of the predictive control algorithm based on the kind of cost function utilized in chapter 5. The stability of complicated nonlinear control algorithms such as the predictive controller is not as easy to trust because of the algorithms complexity. Producers of cars and other automated machinery want to be be sure that their products stay stable and reliable throughout their in-

tended lifespan. But neural networks are nonlinear and thus all the usual stability theory for linear systems will no longer apply. But there are still theory that can be used to show stability for nonlinear systems. One of the most widely used nonlinear stability theorems is the Lyapunov stability theorem. Chapter 6 contains a proof of stability for the predictive controller strategy utilized in chapter 5 based on an already existing proof for a simpler cost function.

Chapter 7 concludes this work with an analysis and commentary of the results achieved and with suggestions for future work in this area.

Appendix A contains a description of the software developed during this work.

1.3 A Brief Introduction to Neural Networks

1.3.1 Complexity and Accuracy

As an alternative to other black box modelling methods, neural networks have become very popular to model difficult and complex parts of real world systems. As an attempt to find an easy inexpensive modelling strategy, but also as a method to get higher modelling accuracy and thus also better control performance.

A neural network is basically a regression function with a special structure. The idea is generally to model the mean value of the activation signal in a human neuron. In the human brain, such neurons exists in large numbers and are connected to each other in very complex ways.

Research has shown that these cells communicate with each other by sending an electric signal through many and long axons and dendrites (the transmitters and receiver lines between the neurons in the brain). The neurons that receive signals from other neurons will then usually themselves begin to send an electric signal to the neurons that they are connected to when a certain conditions are right.

Real neurons send their signals to other neurons as pulses. This is, however, not so practical for a reasonably simple mathematical representation in a computer. It is therefore modelled in a mathematical representation of a neuron as the mean value of the number of pulses sent.

The typical way to model this average pulse throughput for a real neuron resembles the activation function depicted in figure 1.1 although the real function is

much more complicated.

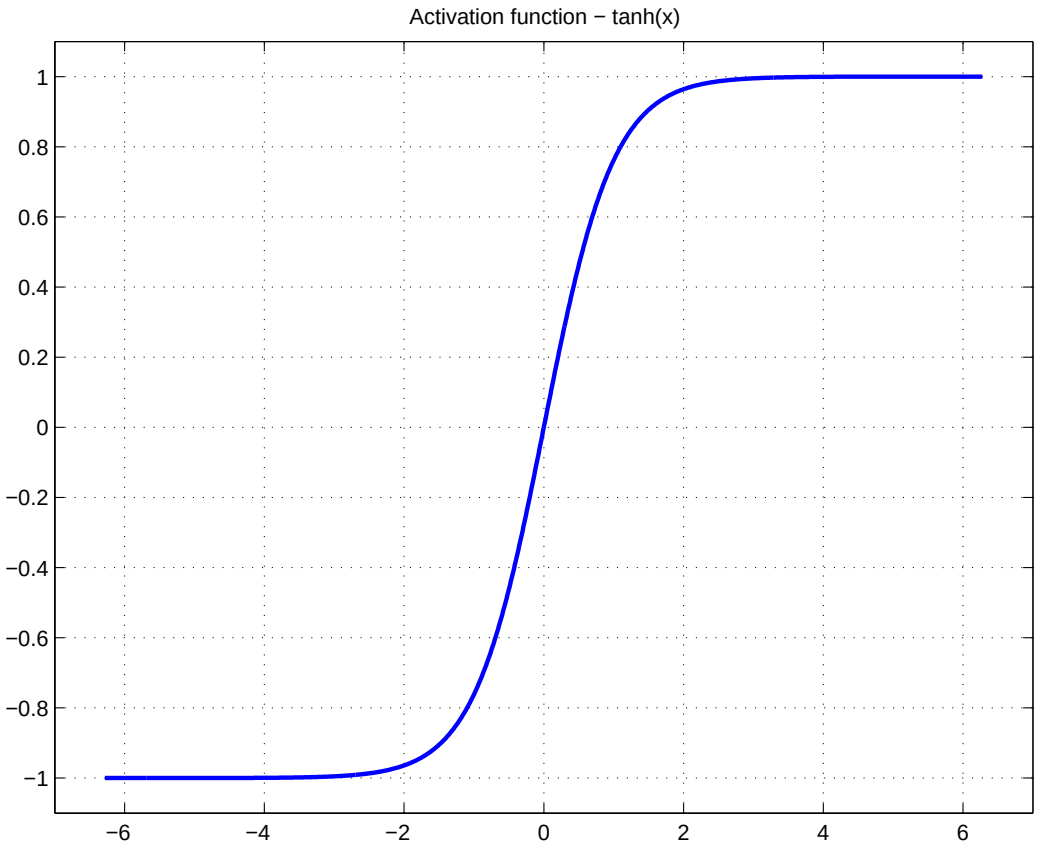


Figure 1.1: Idealized neural activation function

The x-axis represents the mean value of the number of incoming pulses and the y-axis represents the mean value of the number of output pulses coming from the neuron.

The most common kind of neural network utilizing this activation function neuron model is the single hidden layer neural network. Figure 1.2 shows the graphical structure of such a network.

Each neuron works in the way described above utilizing, most commonly, the $tanh(x)$ as the activation function.

A single hidden layer neural network is the type of neural network most often

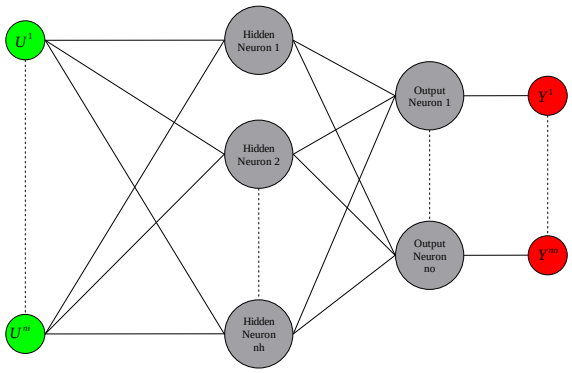


Figure 1.2: Single Hidden Layer Neural Network

utilized because of its ability to approximate any continuous function with any desired accuracy and because it is relatively simple to use.

It is the universal approximation capabilities of neural networks (See for instance [7] and [8]) that make them so attractive for those regression, estimation and modelling purposes of complicated real world systems for which the physics is not known accurately enough. These abilities are described in more detail in chapter 6 in the book [16].

The single hidden layer neural network consist of three layers. An input layer, a hidden layer and an output layer.

The input layer is basically just all the selected inputs. The hidden layer is where all the $tanh$ based neurons are. A neuron in this layer is a nonlinear function of, typically, all the inputs in the input layer.

$$HiddenNeuron_i = ActivationFunction\left(\sum_{j=1}^{N_{inputs}} (WH_{ij} \cdot Input_j) + BH_i\right) \tag{1.1}$$

where

WH_{ij} The hidden layer weight for neuron i and input j.
BH_i The bias for neuron i.

The activation function is usually a sigmoid type function. A sigmoid function is a strictly increasing function that exhibits smoothness and asymptotic properties. Typical examples are.

- tanh(x)
- atan(x)
- sign(x)

The output layer neurons are typically linear.

$$OutputNeuron_i = \sum_{j=1}^{N_{HiddenNeurons}} (WO_{ij} \cdot HiddenNeuronOutput_j) + BO_i \quad (1.2)$$

An engine is a highly nonlinear and complex control object that is very difficult control with the accuracy necessary to meet the ever increasingly demands for lower emissions. This calls for models and control strategies that are more flexible and has a greater ability respectively to reflect reality and to respond in more complex ways to system behavior. The single hidden layer neural network has been chosen as the kind of network used in this work because it is relatively simple both mathematically and with respect to implementation.

1.3.2 Network Training

In the last few years some attempts to utilize neural networks as model and controller subcomponents have been made in the area of engine control and has resulted in papers such as [30], [22], [15], [46], [42], [41], [33], [34], [37], [26], [9], [38] and [6]. It is immediately apparent from the engine control and modelling literature that many people are still utilizing the back propagation algorithm in various versions for neural network training.

The back propagation method has very slow convergence properties (See [16] chapter 6) and is only a first order steepest descent method known from optimization theory. It is mainly used because it is relatively simple to implement, even for multi layer neural networks.

Better methods like the Levenberg-Marquardt algorithm ([27]) have been available since 1963 and have much faster and robust convergence properties.

The training of neural networks very often involves the minimization of a quadratic cost function of the following type.

$$J = \frac{1}{2N}\sum_{i=1}^{i=N}\left(Y_i - \hat{Y}_{i|W}\right)^T\left(Y_i - \hat{Y}_{i|W}\right) + \frac{1}{2N}W^T DW \tag{1.3}$$

Where

- N is the number of input examples to the neural network.
- Y_i are the actual and desired target outputs.
- W are the neural network weights.
- $\hat{Y}_{i|W}$ are the neural network outputs given the current weights.
- D is the weight decay weight matrix, a regularization weight.

The Levingberg-Marquardt training algorithm is an optimization method which is designed specifically to handle quadratic cost functions such as the neural network training kind of cost function in equation 1.3.

The neural network training cost function in equation 1.3 is designed to minimize estimation errors, but also includes the possibility of adding a regularization term (The $\frac{1}{2N}W^T DW$ term in equation 1.3. This regularization term helps avoid local minima at the expense of estimation accuracy. It achieves this by letting the size of the neural network weights add to the value of the cost function. This has the effect, as can be seen from equation 1.3, to make all the small estimation errors less significant compared to the paraboloid in the weight space coming from the regularization term $\frac{1}{2N}W^T DW$. The error surface becomes smoother and many local minima are thus avoided. This also tends to make the neural network more generalizing and thus a better estimator for inputs it did not see during training.

1.3.3 Modelling

The complex physics of an engine makes it very hard to analytically derive a satisfactory model. Many engine modelling people thus turn to neural networks because of their ability to obtain information from a system by utilizing only measured inputs and outputs.

The measured input-output data is used as training examples for a neural network structure and an optimization algorithm is then utilized to minimize the difference

between the networks output and the measured output by adjusting the neural networks free parameters often referred to as weights and biases.

In general this is done on an entire system of some sort with with well defined inputs and outputs and very little physical knowledge about the system is required. But this has the disadvantage of losing information about internal states of the system and of making the networks too large and complex so that the optimization algorithms will have problems finding a proper local minima with a small enough error.

Simplicity and preservation

A different kind of approach and perhaps a better one is to use many smaller neural networks to model engine subsystems. This will preserve many of the interesting internal states of the engine that can be utilized for diagnostics and/or control strategy information. Furthermore the smaller neural networks are much easier to train and will in general perform better than large complicated neural networks. Neural networks with a low number of free parameters, input and outputs are easier to train because they are simpler and thus are more likely to make the error surface for the cost function simpler too.

Another reason for keeping the neural networks smaller and simpler in terms of the number of free parameters (neural network weights, W) is that they will usually tend to over fit the target output data. In other words, that they will relate changes in the inputs to the noise on the target outputs and thus loose a lot of freedom to predict the desired system behavior (See [16] chapter 6). This could also mean that the neural network ability to estimate outputs correctly for inputs it did not see during training might be decreased.

In the papers [41] and [46] neural networks are utilized as physical subcomponent models in an SI engine model. The structure of the model is based on the physical relationship between the various parts of the engine.

The model in that paper is basically split up in three parts: The intake manifold model, the fuel delivery model and the torque model.

The intake manifold model alone contains three different parts that are modelled by neural networks. Two networks for the throttle functions and one for the air mass flow to the cylinder. The equations for this model appears as follows.

$$\begin{aligned} \dot{m}_{at} &= NN_2(\alpha, N)\sqrt{\frac{1}{RT_a}}NN_1(\frac{P_m}{P_0}) \\ \dot{m}_{ap} &= F(NN_3(P_m, N), T_a) \\ P_m &= \int (\dot{m}_{at} - \dot{m}_{ap})\, dt \end{aligned} \tag{1.4}$$

Where

T_a	Is the air temperature.
N	Is the engine speed.
P_m	Is the intake manifold pressure.
$\dot{m}_{at}$	Is the air mass flow from the throttle.
$\dot{m}_{ap}$	Is the air mass flow to the cylinder.
$F()$	Is some dependency on T_a that was not described in the paper.

The main point here is that internal variables like $\dot{m}_{at}$ and $\dot{m}_{ap}$ are preserved and can be used for other purposes too. Furthermore, the neural networks will model more well defined, although complicated, physical relationships and will thus be easier to parameterize(train). This is important because this also makes it simpler to verify whether or not the neural networks generalize well enough.

1.4 Related Work

1.4.1 Virtual Sensors

Not much work has been done on the application of neural networks for engine modelling and control. There are a few papers about neural network based engine control, but most of the engine related papers about neural networks are about the application of neural networks as virtual sensors.

Virtual sensors are, however, also an important and useful subject. Especially in the car industry where the profit margin is very low. If an expensive sensor, like the lambda sensor, can be replaced by some cheaper pressure sensors and an estimation algorithm in the engine controller unit (ECU), then the production cost of each car can be reduced significantly.

This work contains an analysis of the application of neural networks as virtual sensors (Chapter 2) based on pressure measurements in the cylinders of a diesel engine. The neural networks abilities to estimate exhaust O_2 contents, O_2 contents

in the cylinder during combustion, peak pressure and peak pressure location are examined and evaluated.

Recent work on virtual sensing using neural networks encompasses issues like:

- Knock detection. [30]
- Misfire detection. [22], [37]
- Emissions and torque estimation. [15]
- Air fuel ratio estimation from ion current measurements. [33]
- Cylinder pressure and torque estimation. [42], [26]
- Speed prediction. [6]

They all give promising experimental results that clearly show the usefulness of neural networks as virtual sensors. The reduction in sensor costs must be weighed against the cost of the necessary computing power. Neural networks consume a larger amount of processing power. The price of fast CPU's is, however, constantly decreasing and will soon not be a very costly component in the making of a car.

1.4.2 Neural Network Model Based Controllers

Spark advance control

The papers [25] and [34] both apply neural networks as virtual sensors detecting the location of the peak pressure. The location estimate is then utilized to adjust the spark advance in order to control the location of the peak pressure.

The authors of [25] utilize pressure measurements at crank angles 20 degrees apart as inputs to a neural network. The neural network then produces an estimate of the location of the peak pressure. The estimate is then subtracted from the target location of the peak pressure and the error signal is processed by a linear controller. The linear controllers output is the basis of the spark advance control signal.

There is, however, in this case also an adaptive neural network feed forward controller that utilizes engine speed and intake manifold pressure as inputs. This adaptive controller is trained online to make the control signal coming from the linear controller zero. In this way the authors have constructed a very flexible controller that is capable of adapting to wear and other changes in the conditions

influencing the system. The speed of the controller is also increased significantly by adding this feed forward controller.

The authors of [34] utilizes a normalized ion current signal from the ignition coils as a basis for estimating the peak pressure location. The ion current signal is normalized in order to take the effect of fuel additives out of the picture. Fuel additives changes the magnitude of the ion current signal. The location of the peak pressure is strongly correlated with the location of the peak ion current.

However, to further reduce the number of inputs used for a neural network, the ion current signal is also processed by the principal component analysis method. This method utilizes the eigenvectors corresponding to the largest eigenvalues of the data signal covariance matrix and thus reduces the number of data points from the current signal significantly.

The neural network then utilizes these principal components to estimate the location of the peak pressure. A simple PI controller then generates the spark advance timing control signal based on the difference between the target location of the peak pressure and the neural network estimated location.

Mixed Controllers

The paper [9] utilizes a neural network to adapt the forgetting factor in an adaptive control algorithm. Although the authors of [9] utilizes a computer model of an engine instead of real measured data, the idea is a fairly interesting example of how to utilize a neural network to optimize the performance of a control system.

The adaptive algorithm identifies a model describing the relationship between a fuel injection time constant and the normalized air fuel ratio λ. The adaptive algorithms reaction speed to changes in the system is adjusted by a forgetting factor β where $0 < \beta < 1$. The forgetting factor simply weighs the last error by β, the second last one by β^2 and so on.

The authors of [9] then observed that a different optimal value for β exists for different operating points of the engine. The optimal value of β depends on several engine states in a complex way that would be very difficult to describe with physics. A neural network is then trained to output optimal β values as a function of engine speed, manifold pressure and a value called the transiency variable. The transiency variable is defined as the difference between the average manifold pressure over the last 7 engine cycles and the actual manifold pressure. The au-

thors of [9] do however not mention how the optimal values of β have been found.

This is an interesting application of a neural network as part of a controller. It would however have been more useful if the authors had tried this out on a real engine instead of a model. There is also the issue of stability which the authors of [9] do not mention in the paper.

1.4.3 Direct Neural Control

There have not been any attempts, known to the author, of utilizing neural networks directly as controllers for engines. This is a a wealth of opportunities since a neural network represents a most flexible controller that can be shaped to yield any continuous control signal that may be necessary to obtain the desired performance on difficult and complex systems.

The paper [21] shows a very interesting example of how to construct a neural network controller based on Lyapunov stability theory. They assume a very general system description.

$$\dot{x} = f(x, u) \tag{1.5}$$

The idea is to create a positive definite function and then train a neural network as a controller that makes the positive definite function a Lyapunov function for the closed loop. The particular choice in [21] is

$$V(x) = \frac{1}{2}(x - x_d)^T Q (x - x_d) + \frac{1}{2}(u - u_d)^T R (u - u_d) \tag{1.6}$$

Where x is the system state vector and x_d the desired system state vector. u and u_d is control signal and the corresponding control signal for the desired system state respectively. This strongly resembles the cost function of an LQR controller.

The authors of that paper achieves negative definiteness of the derivative of the positive function by using a special cost function for the training of the neural network.

The freedom to choose a positive definite function together with the flexibility of neural networks as a controller is a powerful combination. It gives a broad variety of possibilities for performance requirements since the shape of the Lyapunov function has an influence on the closed loop time constants.

1.4.4 Predictive Neural Network Model Based Control

Nonlinear predictive control utilizing neural networks as models/predictors for engine control is not being utilized in engine control. Even though the excellent performance of this type of controllers on nonlinear systems is evident, see for instance the papers [36] and [47] and the dissertation [31].

The idea here is to train a neural network as a model of the desired system and then minimize the following cost function at each sample time step.

$$\begin{aligned} J(k) = &\sum_{i=N_1}^{N_2} \left(Y(k+i) - R(k+i)\right)^T \left(Y(k+i) - R(k+i)\right) \\ &+\rho \sum_{i=1}^{N_u} \left(\Delta U(k+i)\right)^T \left(\Delta U(k+i)\right) \end{aligned} \tag{1.7}$$

$$\Delta U(k) = U(k) - U(k-1) \tag{1.8}$$

Where $Y()$ is the neural network model output vector, $R()$ is the reference vector, $\Delta U()$ is a vector containing the changes in the control signal and ρ is a constant to weigh the size of the change in the control signal against the size of the *predicted* error.

The cost function is based on input and output values in the future and therefore make it necessary to know the reference N_2 steps in the future and to have a good model of the system for output prediction. A neural network has the potential to be a good model and provide the predicted system output values for a specific set of future control signals.

A minimization, takes place at each sample time step and finds the set of future control signals that minimizes the cost function 1.7.

In the papers [36] and [47] and the dissertation [31] it is obvious how powerful this method of control is for nonlinear systems and it deserves to be examined for engine control purposes.

Furthermore, nonlinear predictive control has been the subject of many attempts to mathematically prove stability. See for instance [47] and [40]. Although many of them have quite a few demanding and unrealistic assumptions, they strongly indicate the stability of this kind of algorithm and the performance is excellent

(See for instance [31] and [39]).

The reason this kind of controller is not yet being widely applied is because of the lack of computing power in the engine control units utilized in the automotive industry these days. But processing power is becoming cheaper and cheaper every month and makes this kind of advanced nonlinear engine control a possibility in the near future.

Chapter 2

Virtual Sensors

2.1 Introduction

Neural networks are flexible mathematical structures with the ability to approximate all continuous mappings with arbitrary accuracy. This is one of the reasons why neural networks also are popular as virtual sensors. Virtual means that the sensors which are made with neural networks have the functionality of real sensors, but do not exist. What is being utilized is the redundancy built into physical systems by virtue of the interdependence of their state variables.

This chapter describes how data from a 2.0 L diesel engine has been acquired and how neural networks have been trained to estimate peak pressure value, peak pressure location, the oxygen levels in the exhaust and the oxygen level in the cylinders.

The experiments described in this chapter is an examination of the ability of neural networks to model and estimate various important signals in an engine in order to replace costly sensors by programs in the ECU which will be much cheaper to produce once the code has been developed.

2.2 Data Gathering

This section describes the data gathering process for the data utilized for the in cylinder peak pressure, peak pressure location and the in cylinder lambda neural network sensor training presented later in this chapter. It is included here in this work to help others, interested in improving the results achieved in this chapter, setup similar experiments.

The data was acquired at the engine test stands in Ford Forschnungszentrum Aachen (FFA) in Germany from a 2.0 L Puma diesel engine. Some data sets for the exhaust O_2 concentration had already been acquired, but they did not cover all the interesting Exhaust Gas Recycling (EGR) levels. The available data was utilized anyway as an initial study and is presented in this chapter.

2.2.1 Training Data

Neural network training requires a great amount of data to properly cover the interesting operating area. The following is a description of the data gathering process, the theory utilized to provide the in cylinder lambda value from other sensor data and of the controller constructed to keep EGR levels constant when going through the chosen speed and torque operating points.

2.2.2 Precautions

The peak pressure was constantly monitored throughout the test. If the peak pressure exceeds 140 Bar then the current operating point is skipped for safety reasons.

2.2.3 In-Cylinder Lambda Measurement

It is currently very difficult if not impossible to measure the lambda value directly in the cylinders of an engine. The in-cylinder lambda value thus has to be estimated from other measurable signals such as the CO_2 concentration in the intake manifold. The following sections describe how to estimate the in-cylinder lambda value based on a simplified chemical combustion equation and the CO_2 intake manifold concentration measurements.

Estimating the Lambda Value in the Cylinder

An expression to estimate the lambda value in the cylinder can be found by taking a look at the air flow process in figure 2.1.

Where

F_{in} is the mass burned fraction in the intake manifold.
F_{ex} is the mass burned fraction in the exhaust manifold.
$\dot{m}_{fuel}$ is the fuel mass flow.
$\dot{m}_{air}$ is the mass flow at port.
$\dot{m}_{egr}$ is the EGR mass flow.

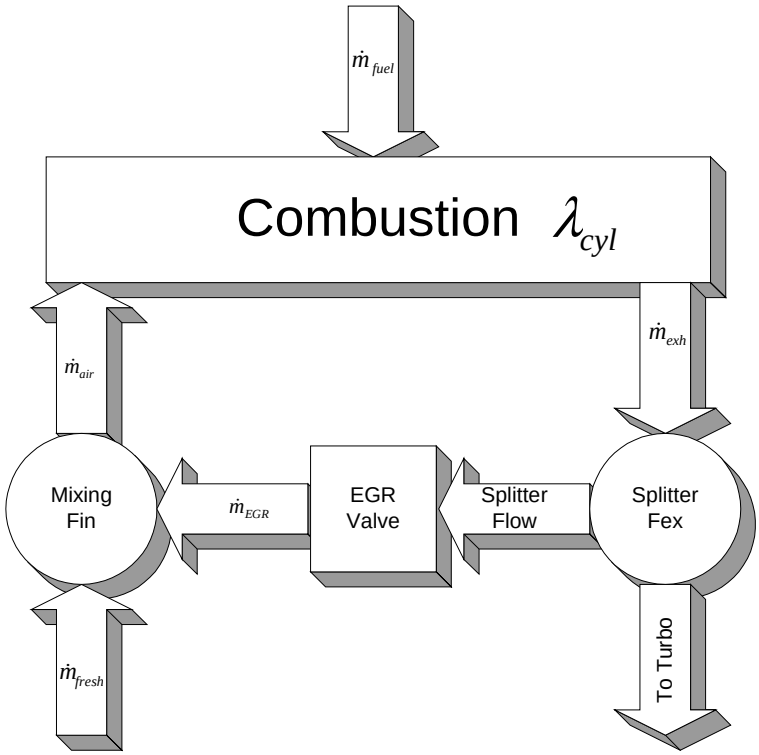


Figure 2.1: Air Flow Process

$\dot{m}_{fresh}$ is the fresh air mass flow.
$\dot{m}_{exh}$ is the exhaust gas mass flow.

It is assumed that ideal mixing of the gasses takes place in both the intake manifold and the exhaust manifold.

The normalized air fuel ratio is defined as follows.

$$\lambda_{cyl} = \frac{(1 - F_{in})\dot{m}_{air}}{X\dot{m}_{fuel}} \tag{2.1}$$

X is the stoichiometric ratio.

Assuming no leaks and utilizing the principle of mass conservation.

$$\dot{m}_{exh} = \dot{m}_{air} \tag{2.2}$$

The mass burned fraction in the exhaust (F_{in}) is the mass of burned gasses divided by the mass of all the gasses in the exhaust manifold.

Equation 2.2 implies that the mass of the burned gasses in the exhaust manifold can be written as the sum of the burned gasses from the intake manifold that go right through the cylinders during combustion, the fresh air that is burned in the combustion itself and the gasses coming from the burning of the fuel itself.

$$\dot{m}_{exh,burned} = F_{in}\dot{m}_{air} + X\dot{m}_{fuel} + \dot{m}_{fuel} \tag{2.3}$$

The total mass of gasses in the exhaust manifold must be the sum of the two parts which are sent into the combustion chamber.

$$\dot{m}_{exh,total} = \dot{m}_{air} + \dot{m}_{fuel} \tag{2.4}$$

So

$$F_{in} = \frac{\dot{m}_{exh,burned}}{\dot{m}_{exh,total}} = \frac{F_{in}\dot{m}_{air} + X\dot{m}_{fuel} + \dot{m}_{fuel}}{\dot{m}_{air} + \dot{m}_{fuel}} \tag{2.5}$$

Solving this equation (2.5) for $\dot{m}_{air}$ yields.

$$\dot{m}_{air} = \frac{1 + X - F_{ex}}{F_{ex} - F_{in}}\dot{m}_{fuel} \tag{2.6}$$

Inserting this into the expression for the normalized air fuel ratio in equation 2.1 then yields.

$$\lambda_{cyl} = \frac{(1 - F_{in})(1 + X - F_{ex})}{X(F_{ex} - F_{in})} \tag{2.7}$$

There were two ways of measuring F_{in} available.

1. by utilizing the two fast oxygen sensors, the NGK and the ETAS sensor.
2. by utilizing the HORIBA CO_2 sensor in the intake manifold.

The first method is less accurate but better for transient measurements. The second is more accurate but not good for transients.

The second method was selected because behavior under transients is not taken into consideration in this experiment and because it is then the most accurate of the two methods.

The CO_2 concentration in the intake manifold was measured utilizing the HORIBA CO_2 probe.

A closer look at the stoichiometric chemical combustion equation is necessary to calculate F_{in} from the CO_2 measurements. The combustion equation has the following appearance assuming stoichiometric conditions.

$$\epsilon CH_y + O_2 + \psi N_2 + mH_2O \rightarrow \epsilon CO_2 + (\frac{y}{2}\epsilon + m)H_2O + \psi N_2 \tag{2.8}$$

For chemical balance, ϵ is required to be:

$$\epsilon = \frac{4}{4+y} \tag{2.9}$$

Where

CO_{2in}	Is the CO_2 concentration in the intake manifold.
y	Is the hydrogen-carbon ratio in the fuel used.
m	Is the $H_2O - O_2$ ratio in the combustion equation under stoichiometric conditions.
ψ	Is the $N_2 - O_2$ ratio in the combustion equation under stoichiometric conditions.

F_{in} can be calculated from the CO_2 measurements by using the following expression derived from the chemical combustion equation 2.8.

$$F_{in} = \frac{1}{4}CO_{2in}(4 + \frac{y}{2} + (4+y)(m+\psi)) \tag{2.10}$$

And the result from equation 2.10 is then utilized in 2.7 for the calculation of the in cylinder lambda value.

2.2.4 Data Points and Signals

Operating Points

The operating points are evenly spread out in a 3D-grid: Crank Shaft Speed, Torque and F_{in} as follows.

- Crank shaft speed at 10 evenly spaced values from idle speed to 3500 rpm.
- Torque at 10 evenly spaced values between maximum torque(including) at the current EGR level and 0 torque(including).
- F_{in} at every 0.1 between 0.0(including) and as high as possible. At least two points, minimum and maximum.

Measured Variables

At each operating point the variables in table 2.1 have been measured including 10 pressure cycles, utilizing the INDISET computer, from all of the already fitted in-cylinder pressure sensors along with the corresponding crank angle. Sampling rate is 1 crank angle degree for adequate detail.

Variables Measured		
Signal Name	Signal Unit	Detail
MAF	$\frac{kg}{s}$	fresh air Mass flow to the engine.
N	$\frac{1}{min.}$	Crank Shaft Speed.
T	Nm	Torque.
O_{2ExNTK}	Vol. %	Oxygen conc. in the exhaust (NGK Sensor).
NO_{xExNTK}	ppm	NOX conc. in the exhaust (NGK Sensor).
O_{2ExHOR}	Vol. %	Oxygen conc. in the exhaust (HORIBA Sensor).
NO_{xExHOR}	ppm	NOX conc. in the exhaust (HORIBA Sensor).
$O_{2InETAS}$	Vol. %	Oxygen conc. in the exhaust (ETAS Sensor).
CO_{2InHOR}	Vol. %	CO_2 conc. in the intake manifold (HORIBA Sensor).
CO_{2ExHOR}	Vol. %	CO_2 conc. in the exhaust (HORIBA Sensor).
CO_{2Amb}	Vol. %	Ambient CO_2 concentration (HORIBA Sensor).
HUM_{Amb}	%	Ambient air humidity
T_{Ex}	C°	Exhaust gas temperature before the catalyst.
P_{In}	Pa	Intake manifold pressure.
T_{In}	C°	Intake gas temperature.
FQ	$\frac{kg}{s}$	Main fuel quantity.
FT	Deg. from TDC	Main fuel timing.
PQ	$\frac{kg}{s}$	Pilot fuel quantity
PT	Deg. from TDC	Pilot fuel timing.
TFF	$\frac{kg}{s}$	Total fuel flow (PLU sensor)
EGV_{Pos}	% closed	EGR Valve position.
EGT_{Pos}	% opened	EGR Throttle position.
VGT_{Pos}	% opened	VGT Throttle position.
P_{Rail}	Pa	Fuel rail pressure.
CO_{Lo}	ppm	CO conc. in the exhaust (Low values CO sensor).
CO_{Hi}	ppm	CO conc. in the exhaust (High values CO sensor).
THC	ppm	Total hydro-carbon conc. in the exhaust.
CH_4	ppm	CH_4 conc. in the exhaust.
$Noise$	DB	Noise level. (AVL noise meter)

Table 2.1: Variables Measured.

2.2.5 F_{in} Controller

In order to be able to make the 3D grid in speed, torque and mass burned fraction (F_{in}), it is necessary to be able to control the mass burned fraction (F_{in}). A controller was constructed to do just that by controlling the EGR valve.

The controller is basically a PI controller with an anti-windup system to prevent saturation and a long recovery time between measurements. It takes the F_{in} reference and utilizes the PI control algorithm to calculate the EGR valve control signal.

The PI controller is switched on when the dynamometer control system is moving the engine to a new operating point and is still on while it is stabilizing. When the dynomometer has stabilized the engine at the desired operating point (speed and torque) the controller is then switched off and the control output is frozen while the dynomometer computer system is measuring the signals in tabel 2.1. The control signal is frozen during the measuring phase in order to keep things steady.

The PI controller has a memory block for holding the controller output. When a certain signal from the dynomometer system indicates that the system is measuring, then a condition in the controller forces the last output, which is normally stored continuously in the memory block, to be the output of the controller. The memory block is during this period naturally not updated.

The controller also has a feature that resets the integrator in the anti windup part in order to prevent long recovery times when the controller output becomes saturated. This works by comparing the output before and after the saturation block and utilize the result of this comparison to reset the integrator.

A SIMULINK block diagram of the F_{in} controller can be seen in figure 2.2.

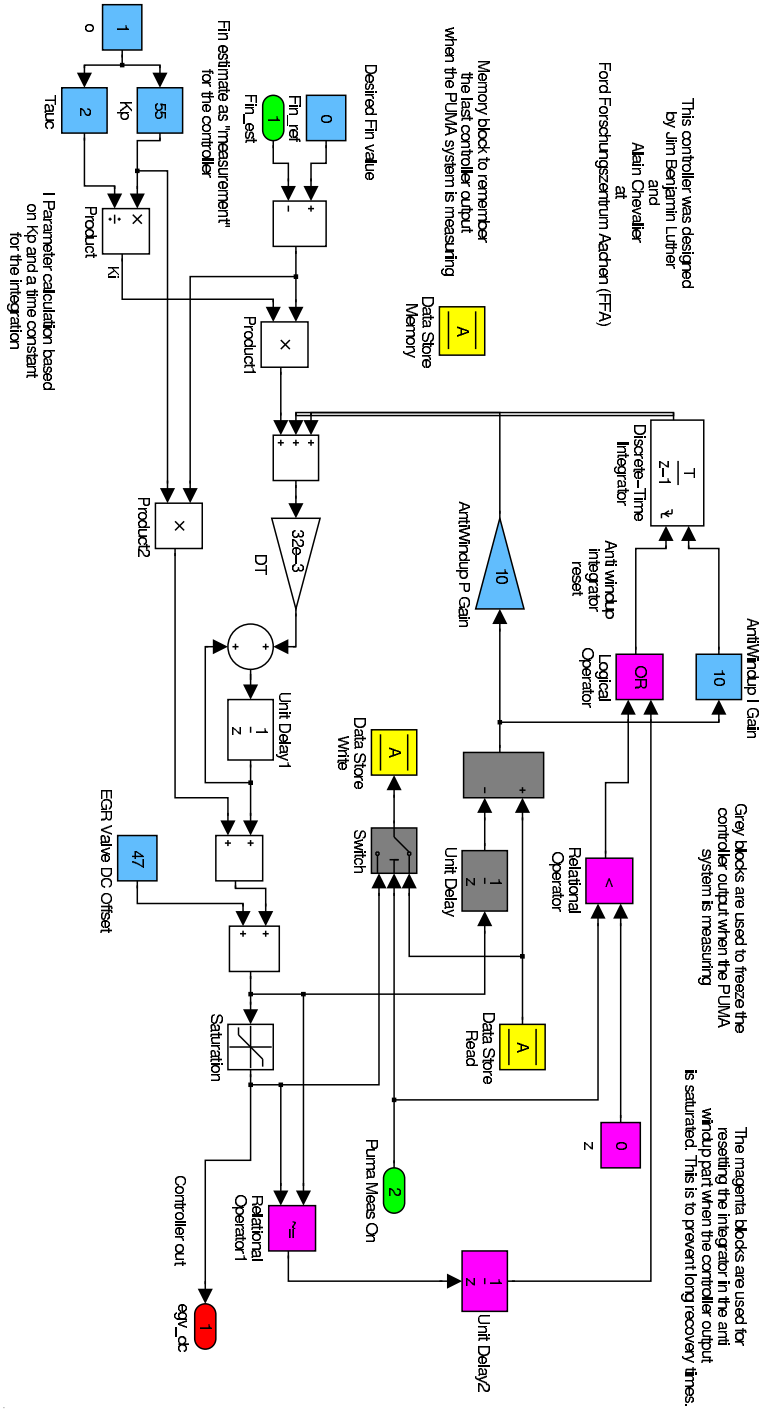


Figure 2.2: F_{in} Controller

2.3 Estimation of O_2 in the Exhaust

2.3.1 Purpose

The idea here is to estimate the O_2 concentration in the exhaust by using various pressure features as an input to a neural network and thereby make it possible to avoid the λ-sensor in a production car, saving money. The O_2 concentration in the exhaust is directly available from the stationary data file from the dynomometer computer. The O_2 concentration in the exhaust is measured by a HORIBA gas analyzer.

2.3.2 Setup

The neural networks are trained with pressure traces recorded earlier by Alain Chevalier and Mathieu Rault at Ford Forschnungszentrum Aachen along with other stationary information listed in table 2.1 for each operating point shown in figure 2.3.

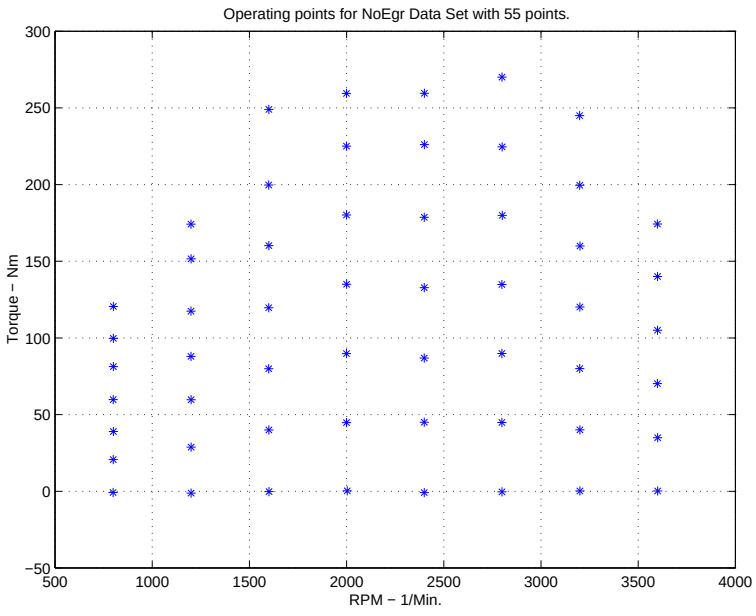


Figure 2.3: Operating Points For Exhaust Tests

There were three data sets with stationary data from all of the operating points shown in figure 2.3 available and EGR had been switched off for two of them. The third data set was recorded with some unknown level of EGR present in the

system.

Network inputs

The 10 pressure trace cycles described in section 2.2.4 form the basis for several inputs, referred to as features, to the neural networks. The features such as the integrated pressure over a cycle are all calculated based on the sampled in cylinder pressure from one of the cylinders.

The features from the pressure traces utilized in this experiment have all been extracted by the MATLAB program extraction.m which performs all the feature calculations and packs the data in a file for the MATLAB program neuralpressure.m. Neuralpressure.m is a MATLAB graphical interface for training the neural networks. The features utilized in this experiment are listed in table 2.2.

1	Maximum pressure.
2	Location of the maximum pressure.
3	Maximum derivative value of the pressure.
4	Location of the of the maximum pressure derivative.
5	Integrated Pressure.
6	IMEP - The area of the pressure volume diagram.
7	Maximum work pr. crank angle.
8	Location of Maximum work pr. angle.
9	Start of combustion.
10	EGR valve pressure difference estimate.
11	Engine speed.
12	Fuel Timing.
13	Intake manifold pressure.
14	Intake manifold temperature.
15	Exhaust manifold temperature.
16	EGR valve duty cycle.

Table 2.2: Exhaust O_2 Feature Table

These features are also the basis for the in-cylinder lambda value neural network virtual sensor experiment in this chapter in section 2.4.

Training

It is however not desirable to just utilize all of the available signals simultaneously as inputs since that would produce a neural network with far too many parameters for practical purposes. The computational load would become too large. The training process would also become increasingly difficult and time consuming with the number of inputs to the network. It will therefore be attempted to keep the number of features inputs to the neural networks as low as possible.

Two different training approaches were applied.

1. The network was trained on a data set where EGR had been switched off and then tested on another data set without EGR.

2. The network was trained on the EGR data set and tested on another data set without EGR.

The second approach will show how well the network generalizes as well as give an idea about how much information the pressure features provide about EGR. There were not enough data sets at the time this experiment took place for training a neural network on that could take the effect of EGR into account. This is therefore not a complete test, but is still very informative.

2.3.3 Results and Discussion

The best combinations of features found during the neural network training was 1,3,7,11 and 15 from the list in table 2.2.

Feature 15 is the exhaust manifold temperature. It is currently an information which is expensive to obtain and this is of course a problem. This particular sensor is in a harsh environment in the exhaust and cannot be expected to be reliable for a long time. It will be a problem for production cars. But it makes a big difference as can be seen from the results on the following pages in figure 2.4, 2.5, 2.6 and 2.7. It is definitely worth considering.

The graphs in figure 2.4, 2.5, 2.6 and 2.7 show the result of the neural network training utilizing the best combination of features.

All the errors mentioned in the following are not relative %, but absolute in O_2 %.

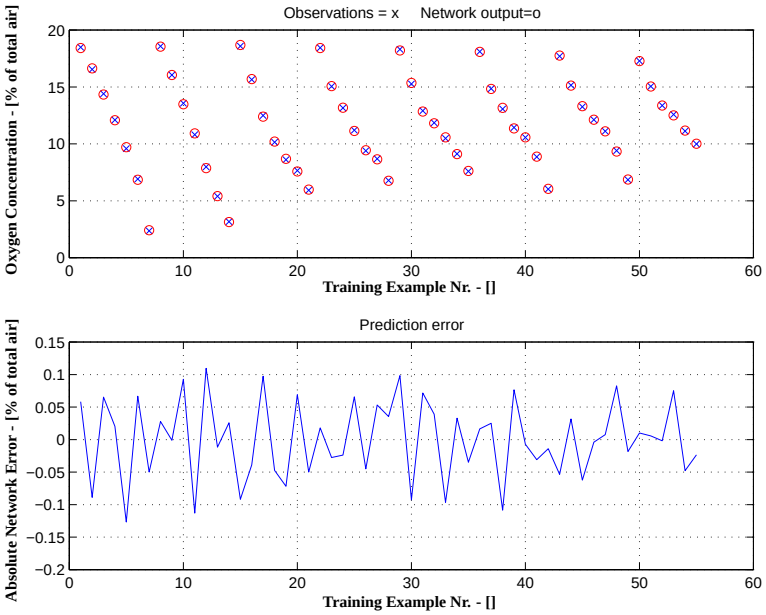
It is immediately obvious from figure 2.4(b) that the neural networks, utilizing inputs 1,3,7,11 and 15 from the feature list in table 2.2, are able to estimate the O_2 with a decent level of accuracy. About ± 0.4 %.

The accuracy clearly decreases when the exhaust temperature is not utilized as an input to the network, which can be seen in figure 2.5(b). The error of the estimation went up to about -3 to +1 %. This is a general trend in all the combinations that have been tried. The accuracy of the estimation always improves significantly when the exhaust temperature is utilized as an input.

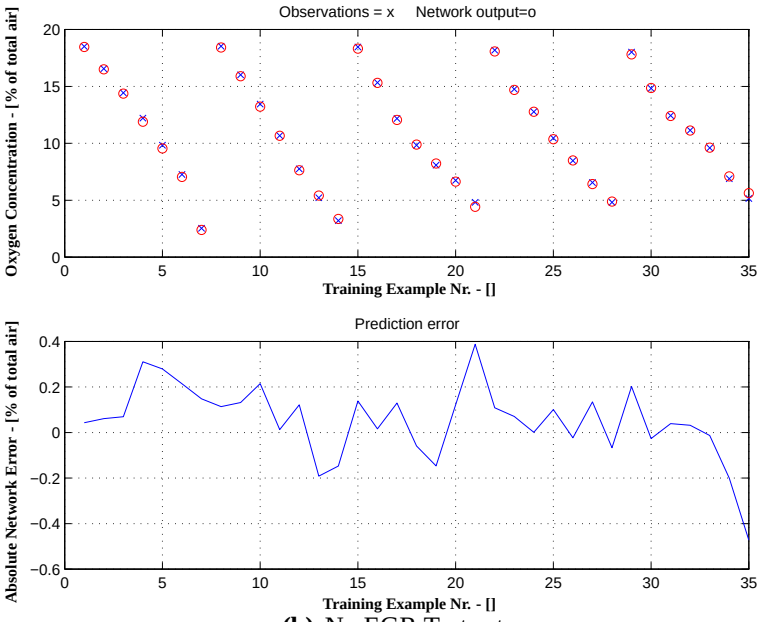
The complexity of the estimation task increases greatly when EGR is introduced into the system. The results are significantly worse, even when the exhaust temperature is utilized as an input. The error, as can be seen from figure 2.6(b), is about -1 to +2.8 % with an exhaust temperature input. It becomes much worse without the exhaust temperature, see figure 2.7(b). The error is then about -3 to +5 %.

The reason for the reduced estimation accuracy when EGR is introduced into the system could be lack of EGR related information in the features listed in table 2.2. Furthermore, the in-cylinder pressure sensors utilized in the diesel engine at FFA are subject to thermal shock effects which can lead to significant errors in the calculation of the pressure features and thus poor correlation between the inputs to the neural network and the target output.

No EGR training, no EGR testing - Utilizing The Exhaust Temperature



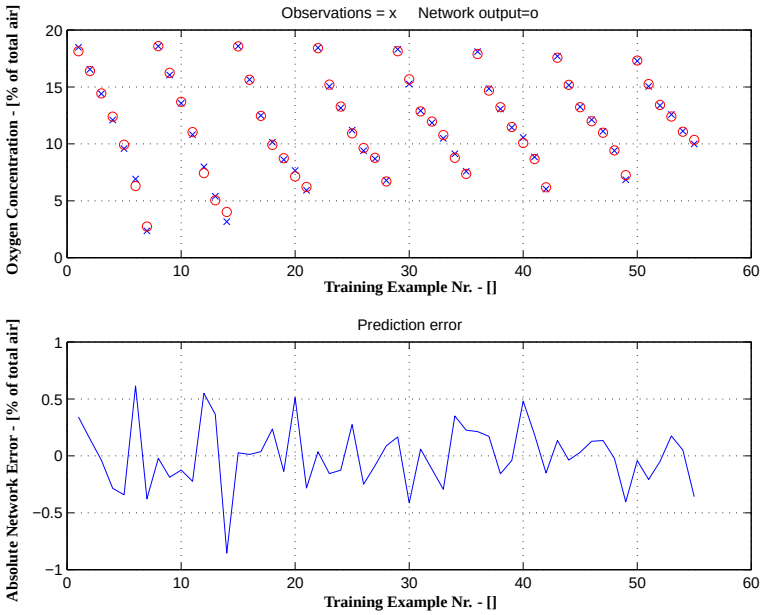
(a) No EGR Training set



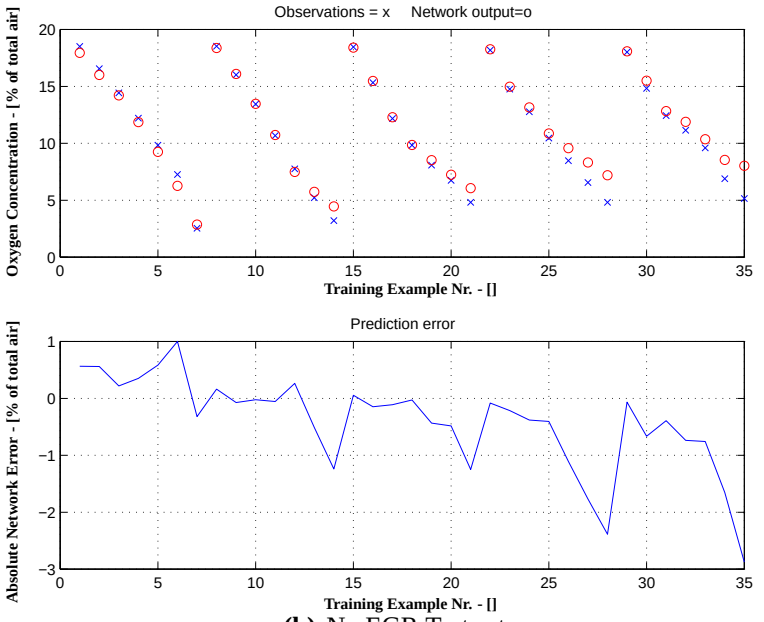
(b) No EGR Test set

Figure 2.4: No EGR Training and Test Results - With Exhaust Temperature

No EGR training, no EGR testing - Not Utilizing The Exhaust Temperature



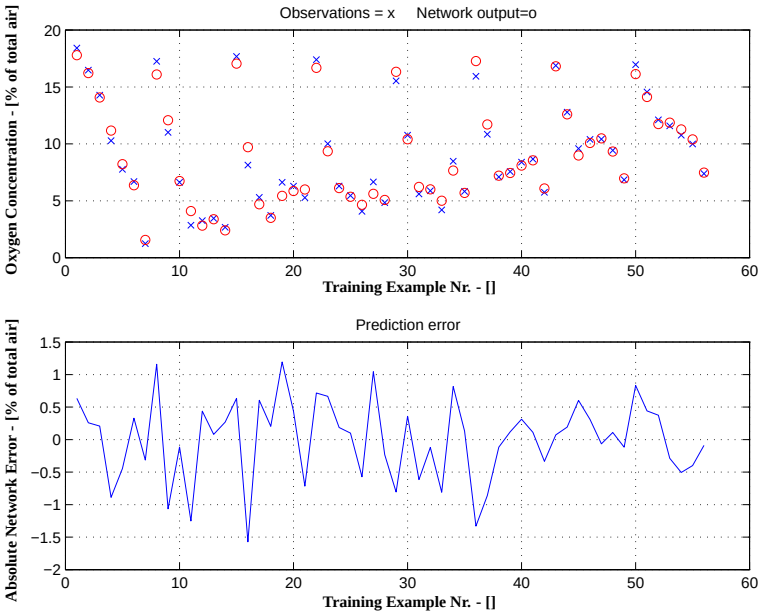
(a) No EGR Training set



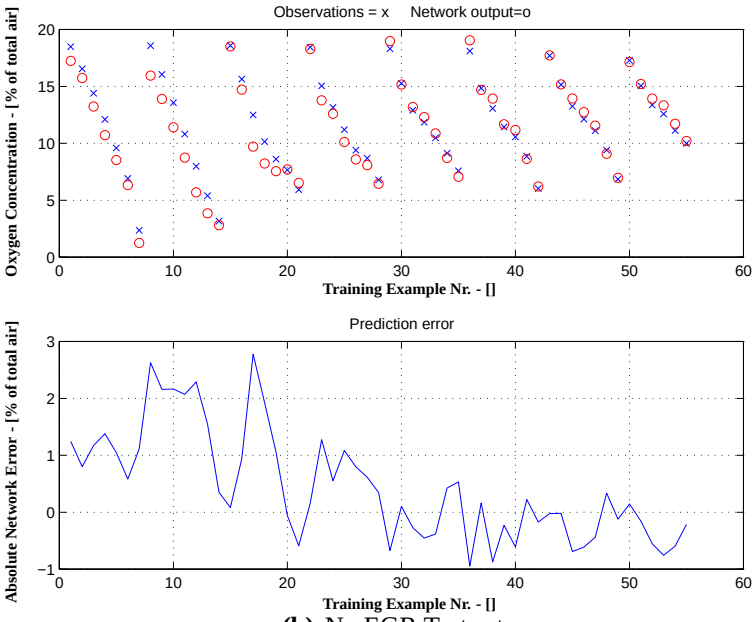
(b) No EGR Test set

Figure 2.5: No EGR Training and Test Results - Without Exhaust Temperature

EGR training, no EGR testing - Utilizing The Exhaust Temperature



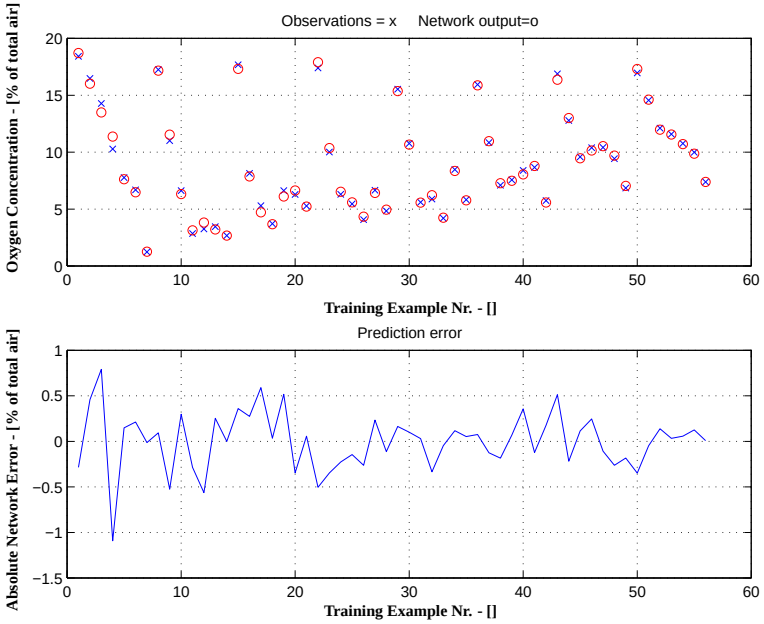
(a) EGR Training set



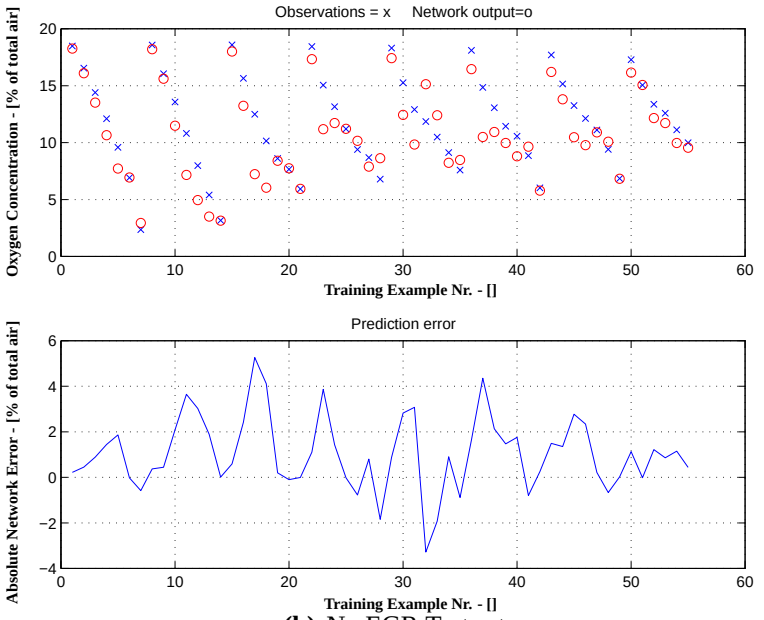
(b) No EGR Test set

Figure 2.6: EGR Training and Test Results - With Exhaust Temperature

EGR training, no EGR testing - Not Utilizing The Exhaust Temperature



(a) EGR Training set



(b) No EGR Test set

Figure 2.7: EGR Training and Test Results - Without Exhaust Temperature

Maximum pressure.
Largest pressure rate.
Integrated pressure.
IMEP - The area of the pressure volume diagram.
Maximum work pr. crank angle.
RPM.
Throttle plate angle.
Intake manifold pressure.

Table 2.3: In-Cylinder Neural Network Estimation Inputs

2.4 Estimation of Air Fuel Ratio in the Cylinder

2.4.1 Purpose

The intention of this experiment is to utilize a neural network to estimate the normalized air fuel ratio (the lambda value) in the cylinder. The in-cylinder lambda value is calculated by equation 2.7.

2.4.2 Setup

The features utilized as a basis for the choice of inputs in this experiment are listed in table 2.2 in the section about the estimation of the exhaust O_2 concentration.

However, not all the available inputs in the table can or should be utilized when experimenting with various neural network input combinations since some of them are directly related to the target output which is the in-cylinder lambda value. The signals not normally available in production cars will also be avoided. The CO_2 measurement in the intake manifold can naturally not be used since this is what the λ value in the cylinder is calculated from and thus directly related. It is however also because it is not measured in a production car. The signals in table 2.2 satisfying these requirements are called the feasible feature and are shown in table 2.3.

The feasible features will be utilized in various combinations as inputs to a neural network. It will be attempted to find combinations that do not require any new sensors since that would ruin the idea of utilizing the neural network as a virtual sensor.

A single hidden layer neural network is trained to estimate the air fuel ratio concentration in the cylinders as calculated from equation 2.7. The neural network is trained on one half of the data and tested on the other half. Each half is is made from every other operating point in the data set in order to ensure that the network is trained and tested on the full range of EGR levels and operating points.

The oxygen concentration in the exhaust, which is used to calculate the mass burned fraction in the exhaust, is measured with an NGK combined O_2 and CO_2 sensor. The HORIBA gas analyzer had some unfortunate fallouts that would exclude too many operating points.

The NGK sensor, however, had a peculiar nonlinear behavior that was observed by making a characteristics curve for the sensor. This is compensated for by utilizing the characteristics curve shown in figure 2.8.

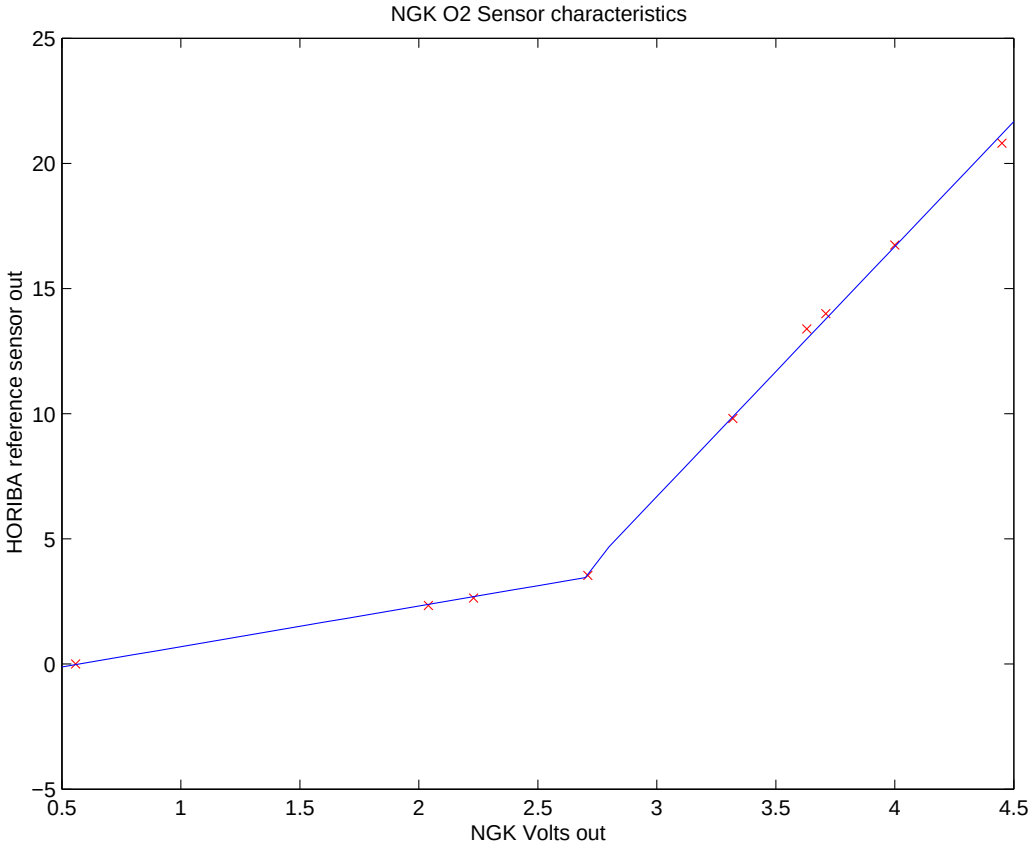


Figure 2.8: NGK Oxygen Sensor Characteristics

2.4.3 Results

The best neural network for estimating the in-cylinder air fuel ratio was obtained by utilizing the pressure features and engine signals listed in table 2.3.

The graphs in figure 2.9 and 2.10 illustrates the performance of the neural networks on the training and test data sets. Figure 2.9 contain the neural network training and test set absolute errors and figure 2.10 contains the neural network test set error relative to the measured data.

2.4.4 Discussion

Figure 2.9(a) shows that the results are for the most part quite good except for some specific point where it looks like there is a problem with the test set error. The error reaches ± 0.4 at some points and this indicates that there are problems with the training material. Some of the inputs examples are not correlated properly with the target outputs. The same can more easily be seen in the test set relative error plot in figure 2.10. The graph indicates an accuracy of about $\pm 38\%$ which is not good enough compared to the computational effort used to calculate the estimate.

The large training and test set errors are most likely to be caused by noise in the pressure signal coming from thermal shock and the sensors precision. The noise can cause severe distortions in the calculated pressure features and thus make the correlation between the inputs and the target output obscure. Especially features such as the integrated pressure and the area of the pressure volume diagram since they will integrate the error. This will make it impossible to obtain a decent accuracy with regression models such as the neural networks utilized in this work.

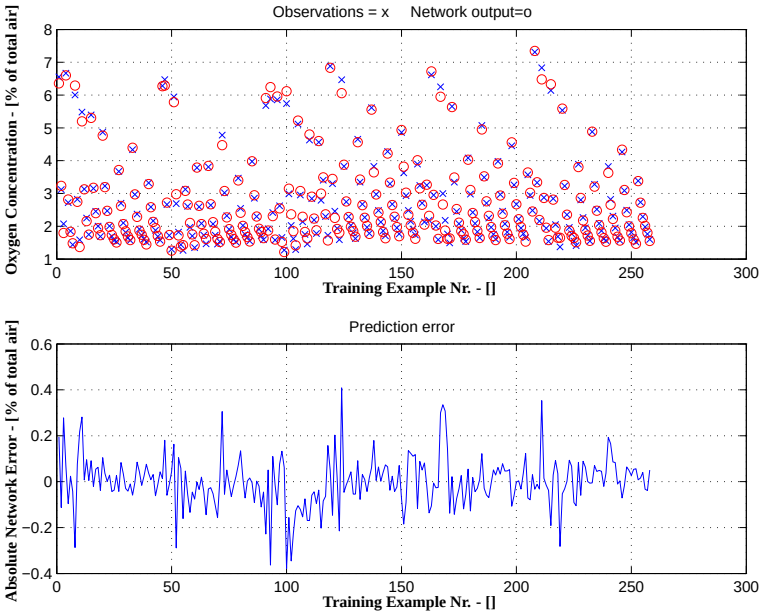
It is likely that this can be improved by going through the whole sensor setup at the engine test stand and make sure that everything works well. It is suspected that some of the sensors utilized in the data gathering process for the virtual sensor experiments in this chapter are a little faulty or perhaps just not good enough for this kind of modelling (Lack of precision, thermal shock etc.). It is very important to make sure that the sensors recording training data are reliable and that they give at least the same level of performance throughout the whole data gathering session. Preferably very accurate sensors. Last, but not least, to keep the noise level as low as possible at all times or at least constant.

But it may also be that the features calculated from the pressure traces are not the right ones to use. One of the other things that could be tried is to utilize pressure

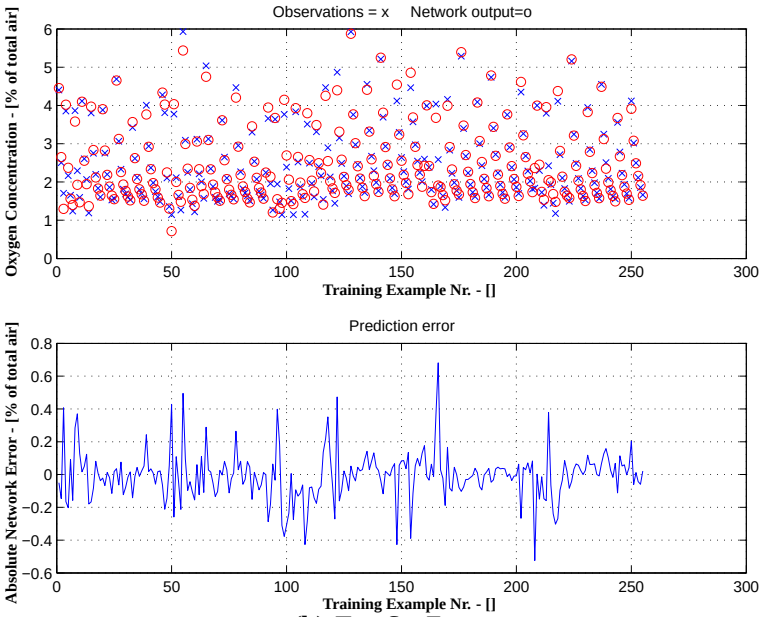
values at predetermined crack angle degrees as inputs to the neural network. Perhaps in combination with some of the information available from the ECU.

Chances are that the networks can do better than this if given some better training material or if better features are extracted from the pressure traces. It is not easy to say exactly which features would be the best ones.

In Cylinder Lambda Estimation - Absolute Errors



(a) Training Set Errors



(b) Test Set Errors

Figure 2.9: Absolute Training and Test Set Errors

In Cylinder Lambda Estimation - Test Set Relative Errors

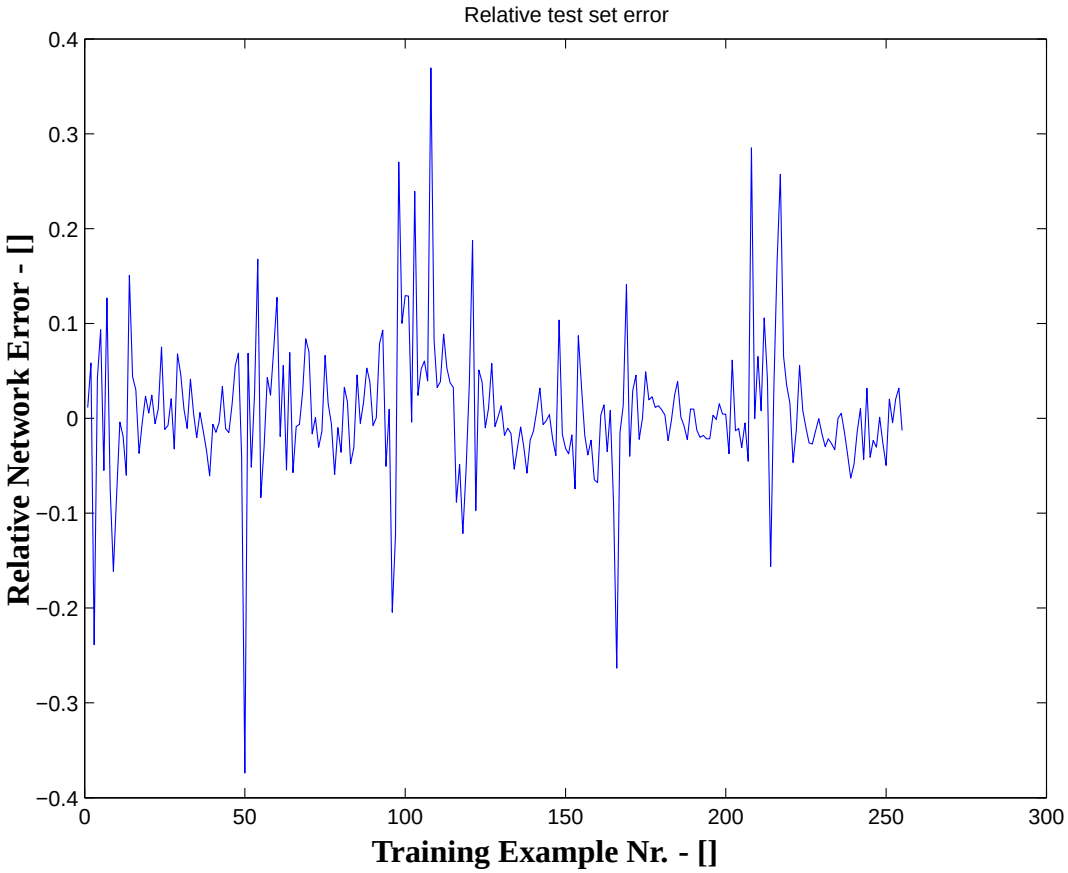


Figure 2.10: Test Set Relative Error

2.5 Estimation of Peak and Peak Pressure Location

2.5.1 Purpose

The intention is to train a neural network to estimate the peak pressure value and the location of the the peak pressure based on a few samples of the pressure trace. Furthermore, to reduce the sampling frequency of the cylinder pressure and still achieve good accuracy. The purpose of obtaining the peak pressure and its location is to be able to balance the cylinder pressure in order to minimize engine vibrations. The pressure could for instance be sampled at every 6 degrees which would be practical since this interrupt can easily be generated by the flywheel for the ECU.

2.5.2 Setup

Many of pressure traces have been recorded on the PUMA 2.0 L engine in FFA's engine test stand. The pressure traces have been recorded at various operating points spanning a grid covering the entire operating area for the engine.

This is not only a grid in speed and torque but a 3D grid in speed, torque and mass burned fraction. The last number is just the fraction of burned gases in the intake manifold.

These pressure traces will form the data material for the following attempts to make a neural network estimate the peak pressure and peak pressure location with as good an accuracy as possible.

Creating data sets

The recorded pressure data contains many more points than needed to train the network, so the pressure data is decimated to smaller data sets containing pressure data for every 6 degrees corresponding to the distance between the teeth on the flywheel.

The training points are chosen at the crank angle degrees shown in table 2.5.2 with 180 being the top dead center.

10 Cycles from each of the 100 operating points plotted in figure 2.11. Which adds up to a total of 1000 points. These points are divided in to two sets. A training set and a test set of 500 points each.

Pressure Training Points								
156	162	168	174	180	186	192	198	204
Crank Angle Degrees								

Table 2.4: Operating Points.

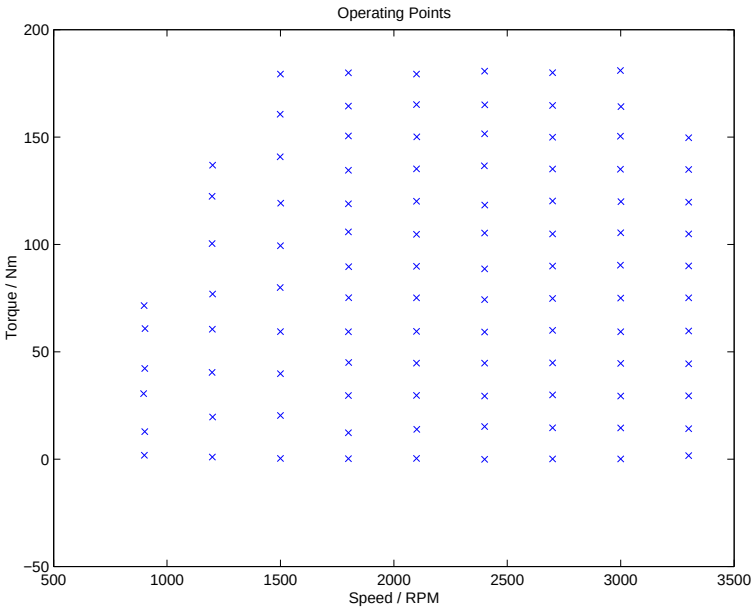


Figure 2.11: Operating Points

2.5.3 Results

Two different neural network configurations were used to estimate the peak pressure and the peak pressure location.

- Two single output neural networks estimating one feature each.
- One double output neural network estimating both features.

The reason for this is that the neural network using two outputs did not seem to be able to estimate the features accurately enough. The single output neural networks performed much better, but this was not immediately clear at first.

2.5.4 Discussion

The two single output neural networks clearly seems to perform much better than the single neural network with two outputs. It is easy to think that the performance of the two output neural networks could be improved by adding more hidden neurons to add more flexibility (Free parameters). This is however basically also what is done by using two output neural networks instead of only one since a two output neural network will have more free parameters than a single output neural network with the same number of inputs and hidden neurons.

Furthermore, adding more hidden neurons did not improve the performance of the two output neural network significantly. The best performance compared to the complexity (number of free parameters) was achieved by single output neural networks as those utilized to produce the results in figure 2.12 and 2.13. Single output neural networks are also easier to train since the complexity of the neural network error function is much simpler and thus do not have as many local minima.

The neural network estimation errors should be evaluated from the test set results since this reflects more accurately how the neural network will respond to data it has not seen before. Taking a look at figure 2.12(b) and 2.13(b), it is seen that the peak pressure estimation errors of the single output neural networks are around -2 to +3 bar at a pressure sampling resolution of 6 crank angle degrees. The peak pressure location estimation errors of the single output neural networks are around -10 to 20 degrees.

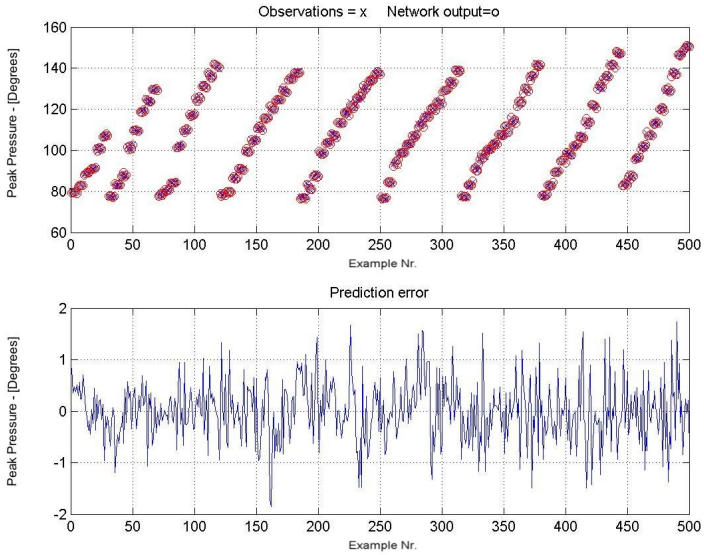
The large peak pressure location estimation errors comes from some strange outliers in the peak pressure location train and test set errors and it is uncertain why they are there and why they are this large. Figure 2.12(a) and 2.12(b) seems to indicate that maybe some of the data points are erroneous (For instance, the points near example nr. 320, 375 and 450) perhaps due to a temporary error in the sensor or some strange effect in the engine. Those points do not seem to follow the pattern of the other points.

Perhaps these data points should be removed for future experiments or new data should be obtained for training. This will probably give much better results.

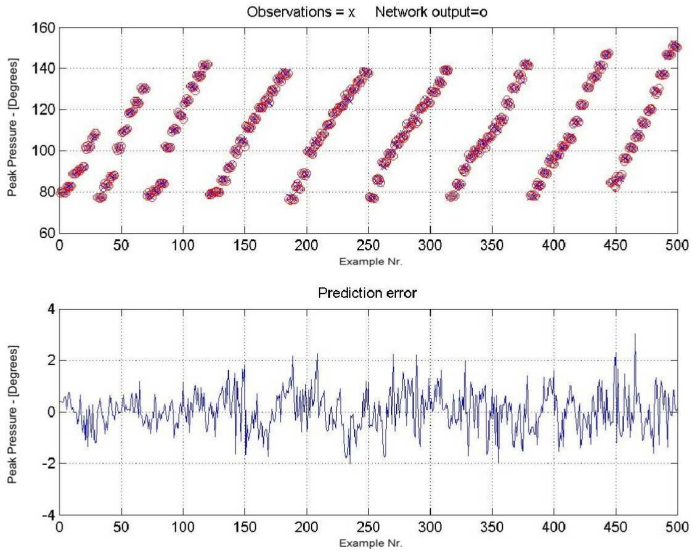
Figure 2.14 and 2.15 both clearly show that the double output neural network does not perform as well as the two single output neural networks. This can possibly be attributed to the more complex neural network error function for a multi output neural network. This produces more local minima and thus makes the neural

network training more difficult because the neural network weights are initialized with random values before training. The path in the weight space walked by the iterative training algorithm can then end up in many non-optimal places.

Peak Pressure Results - Single Output Networks



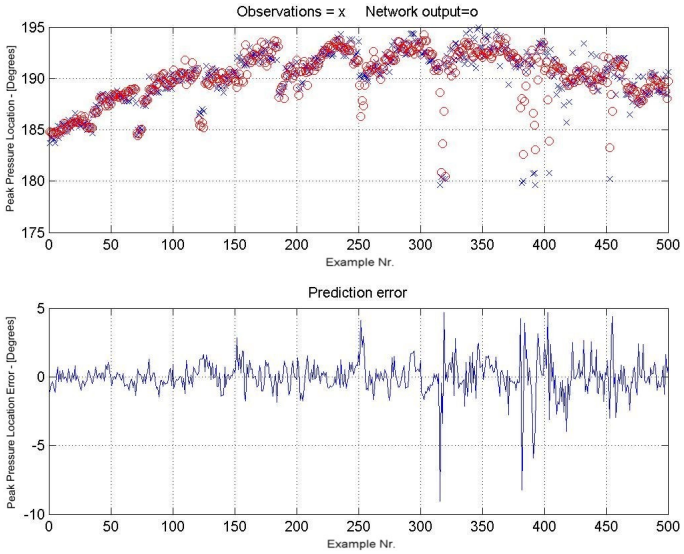
(a) Peak Pressure Training Set Results - One Output



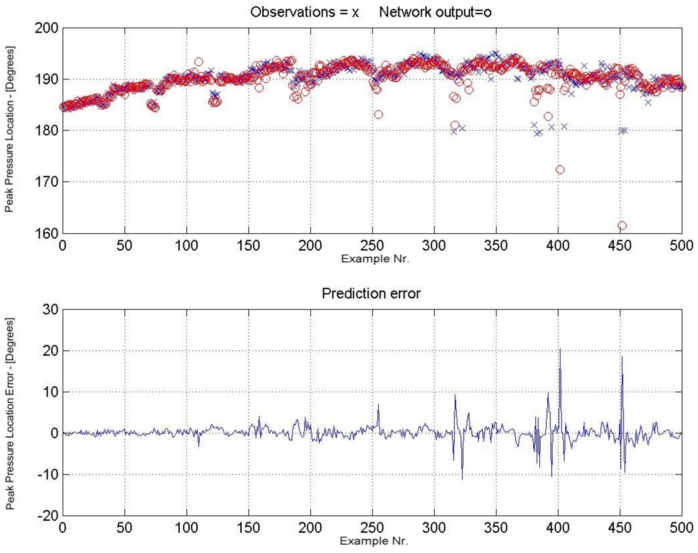
(b) Peak Pressure Test Set Results - One Output

Figure 2.12: Peak Pressure Results - One Output

Peak Pressure Location Results - Single Output Networks



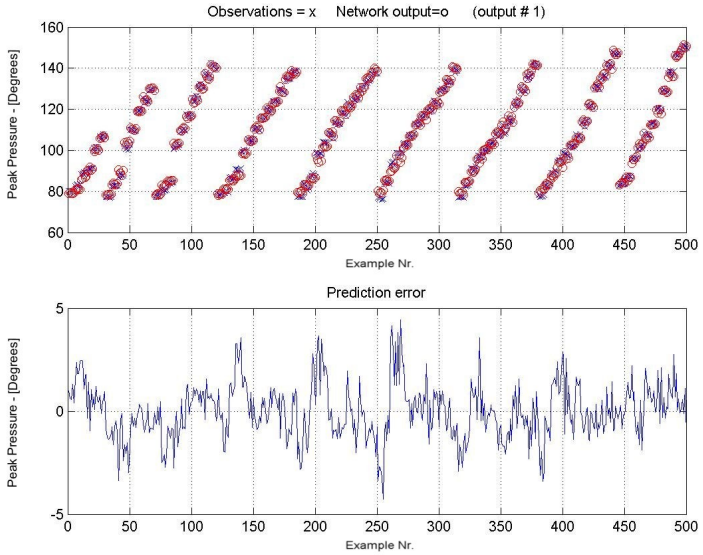
(a) Peak Pressure Location Training Set Results - One Output



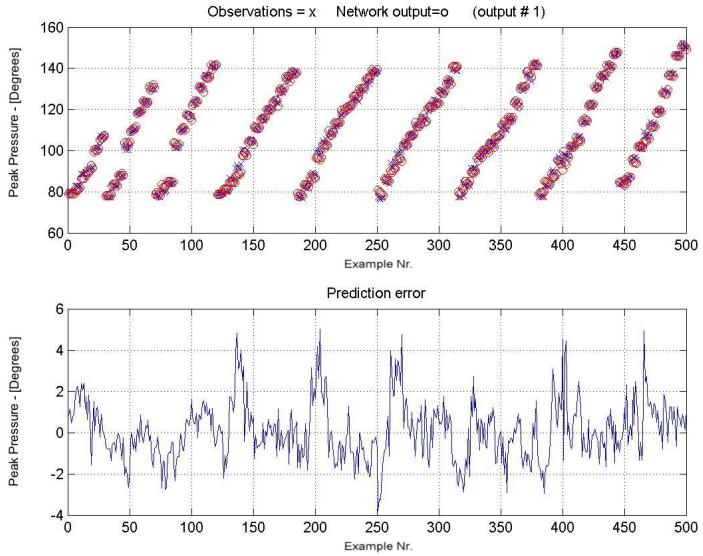
(b) Peak Pressure Location Test Set Results - One Output

Figure 2.13: Peak Pressure Location - One Output

Peak Pressure Results - Two Output Network



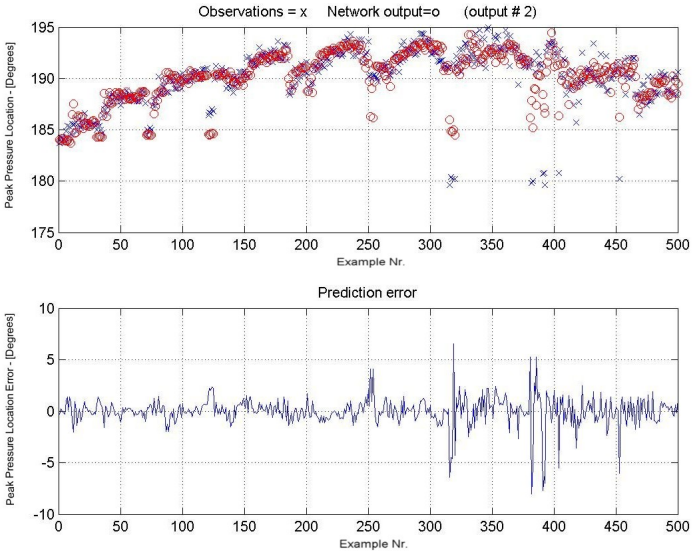
(a) Peak Pressure Training Set Results - Two Outputs



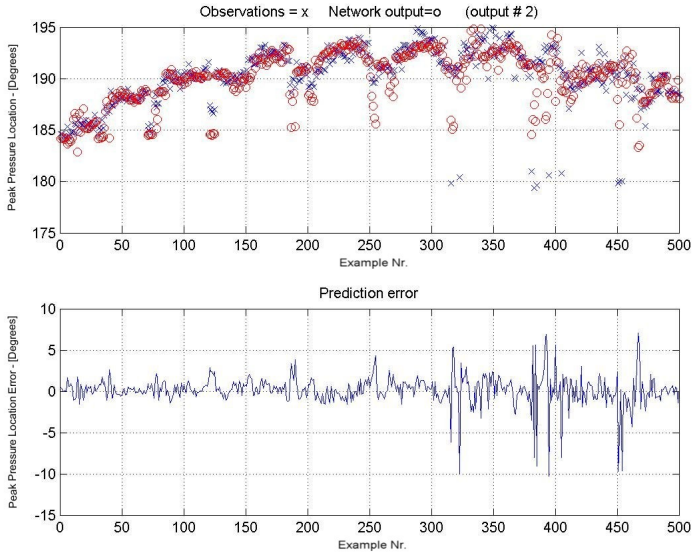
(b) Peak Pressure Test Set Results - Two Outputs

Figure 2.14: Peak Pressure Results - Two Outputs

Peak Pressure Location Results - Two Output Network



(a) Peak Pressure Training Set Results - Two Outputs



(b) Peak Pressure Test Set Results - Two Outputs

Figure 2.15: Peak Pressure Location Results - Two Outputs

2.6 Conclusions

Neural networks has been tested as virtual sensors based on in-cylinder pressure signals from a diesel engine in chapter 2. Several different kinds of virtual sensors were tried. Those were.

1. Exhaust O_2 concentration.
2. In-Cylinder Air Fuel Ratio.
3. Peak Pressure and Peak Pressure Location.

2.6.1 Exhaust O_2 Concentration

The neural network exhaust O_2 concentration estimation was quite good when trained and tested on the no EGR data sets (EGR was turned off during the sampling of these data) and the exhaust temperature was utilized as an extra input to the neural network (see figure 2.4 and 2.5). The largest test set error was in that case about -0.4 %[O_2]. The largest O_2 concentration estimation error increases greatly to about -2.8 %[O_2] for the test set when the exhaust temperature is not utilized.

The neural network virtual sensors were much more difficult to train when EGR was introduced and it seems like the exhaust temperature does not help improve the result in this case (see figure 2.6 and 2.7). The largest O_2 concentration estimation error for the test was about 2.8 %[O_2] numerically when utilizing the exhaust temperature as a neural network input and about 4.5 %[O_2] when the exhaust temperature was not utilized as a neural network input.

The above mentioned errors are peak errors for the test data sets and the overall performance of the neural network exhaust O_2 concentration sensors is ok although not yet good enough to replace a Lambda sensor. The problems are believed to be due to the large sensor noise coming from thermal shock which makes some of the pressure features extracted from the pressure data very inaccurate and thus poorly correlated with the target output.

2.6.2 In-Cylinder Air Fuel Ratio

The neural network in-cylinder air fuel ratio virtual sensor also had some problems with a maximum test set error of about $\pm 38\%$. Those maximum errors are only in a few peaks and the overall picture is not that bad, although still not good enough to replace a lambda sensor. The test set relative errors can be seen in figure

2.10. The error is for the most part smaller than $\pm 10\%$.

The virtual in-cylinder air fuel ratio sensor neural network was in this case also trained utilizing the same in-cylinder pressure features as for the exhaust O_2 concentration. And those features are due to the pressure sensors noise level not good enough to provide the necessary correlation level for good virtual neural network sensor training.

2.6.3 Peak Pressure and Peak Pressure Location

The peak pressure neural network virtual sensor performed fairly well with an error of about ± 2 bar for the most part. A single spike in the error reaches 3 bar. The pressure was sampled at a resolution of 6 crank shaft angle degrees which still makes the remaining error somewhat large compared to the relative small changes (about 8 bars in the area near the pressure peak) in the pressure samples at a resolution of 6 degrees. The error could however not be made smaller and it is believe that this is because of sensor noise.

The peak pressure location neural network virtual sensor was not successful with the available data for this work. The maximum test set error reached 20 degrees and is unacceptable. The data was full of odd outliers which the author has not been able to explain by anything else than sensor errors. It is however believed that the results can be improved by either acquiring new data sets or by cleaning up the existing ones, but the sensors noise is still a problem here too and should be made smaller.

2.6.4 Overall Virtual Sensor Conclusions

The virtual sensor experiments looks promising, but sensor noise from current pressure sensors available for mass production appears to be a problem. To much data is lost in noise or disfigured in a way that makes it extremely complicated to find a suitable set of input signals and network structure.

Another possibility is that the in-cylinder pressure is physically just too complicated for this kind of neural network setup. Advanced signal processing of the in-cylinder pressure signal and a detailed mathematical model of the in-cylinder pressures behavior might be necessary in order to improve the results.

The author believes that better in-cylinder pressure sensors and/or a physical/mathematical study of how the in-cylinder pressure behaves the way to improve the neural network virtual sensors.

Chapter 3

Dynamic Neural Network Model Training

3.1 Basic Dynamic Neural Network Model Training

3.1.1 Dynamic Neural Networks

The training of a neural network as a discrete system model is the almost the same as training the neural network to be a one step ahead predictor. A discrete system model and a predictor both have to be able to calculate the output one step ahead from any given time, but the discrete system model has to be able to continue to predict the system output correctly given its previous predictions as inputs along with the systems external inputs.

The training of a dynamic neural network is however usually setup as a one step ahead predictor with the intention of training the neural network to be so good a one step ahead predictor that it also will be able to continue to make accurate predictions even when it continues utilizing current inputs and previous predictions.

The typical setup for a single hidden layer neural network appears in general as in figure 1.2. The network is made dynamic by feeding back the neural network outputs time delayed and also possibly some or all of the inputs time delayed. Furthermore, the network is trained to predict the system outputs for the next time step. Figure 3.1 illustrates graphically how this works.

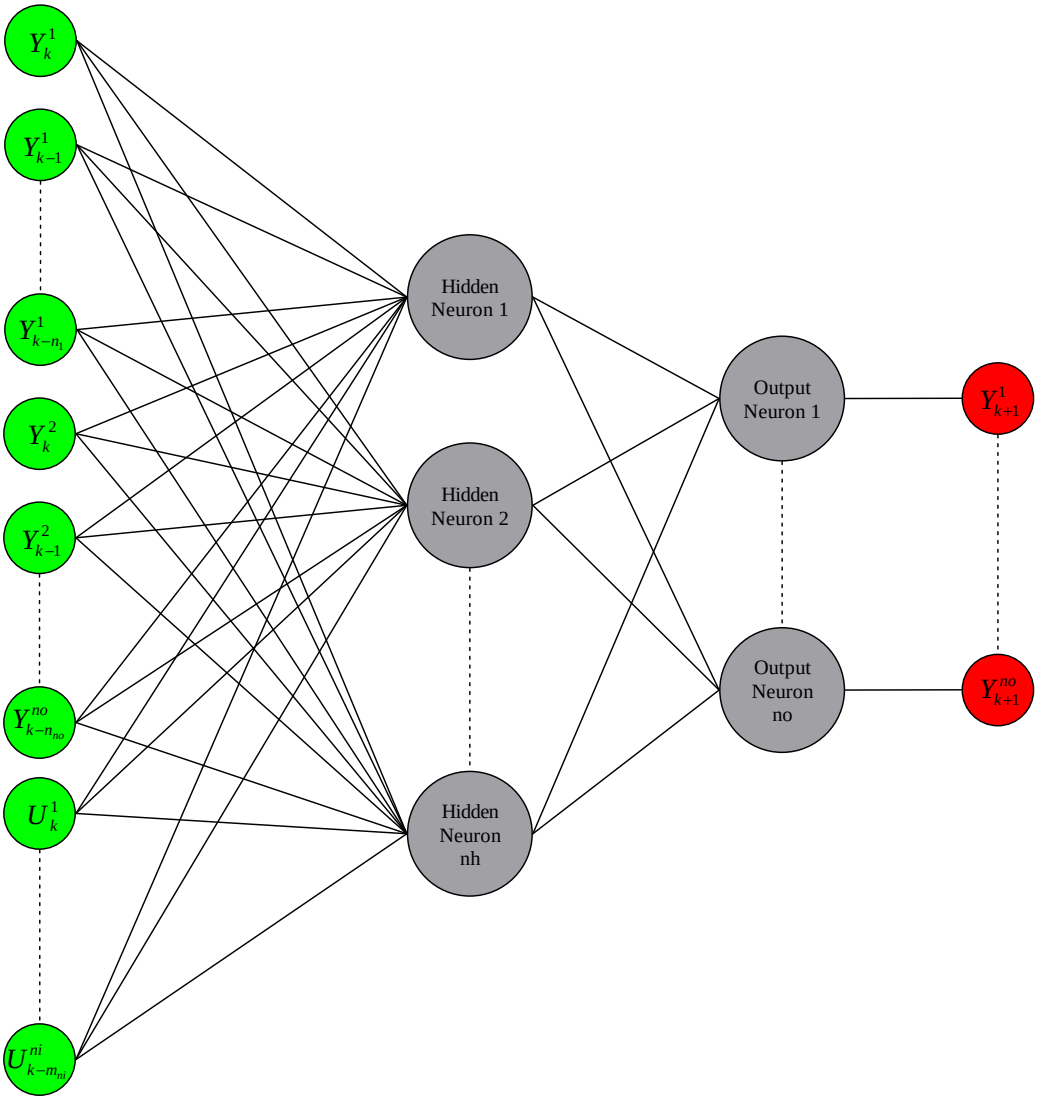


Figure 3.1: Dynamic Neural Network Setup

3.1.2 Dynamic Neural Network Training

The basic principles of neural network training is explained in section 1.3.2. The training of a dynamic neural network is basically done in the same way. The difference is that the input data and target output data are measurements from a dynamic discrete system which is assumed to be describable by the following discrete system equation.

$$Y_{k+1} = F(Y_k, \cdots, Y_{k-n+1}, U_k, \cdots, U_{k-m+1}) \ , \ F \in \mathcal{C} \tag{3.1}$$

$$Y_k = \begin{bmatrix} y_k^1 \\ y_k^2 \\ \vdots \\ y_k^{n_{out}} \end{bmatrix} \ , \ U_k = \begin{bmatrix} u_k^1 \\ u_k^2 \\ \vdots \\ u_k^{n_{in}} \end{bmatrix} \tag{3.2}$$

Where

n_{in}	Is the number of inputs to the system modelled.
n_{out}	Is the number of outputs from the system modelled.

The measured data has to be arranged in a specific way in order to provide the neural network training algorithm with inputs and outputs that belong together with respect to time. All target output training data must be one time step ahead of the corresponding input data and in the same order as shown in figure 3.1.

The standard Levenberg-Marquardt optimization method can be utilized to train the neural network as a one step ahead predictor. This optimizing algorithm is utilized by the `NetworkTrainer` class documented in appendix A.7.

The input and output set matrices given to the `NetworkTrainer` class will look as follows when set up correctly for dynamic neural network training. Each column represents one data set.

Input set matrix:

$$\begin{bmatrix}
y^1_{maxd} & \cdots & y^1_{maxd+ndft-1} \\
y^1_{maxd-1} & \cdots & y^1_{maxd-1+ndft-1} \\
\vdots & & \vdots \\
y^1_{maxd-n_1+1} & \cdots & y^1_{maxd-n_1+1+ndft-1} \\
y^2_{maxd} & \cdots & y^2_{maxd+ndft-1} \\
\vdots & & \vdots \\
\vdots & & \vdots \\
y^{n_{out}}_{maxd-n_{n_{out}}+1} & \cdots & y^{n_{out}}_{maxd-n_{n_{out}}+1+ndft-1} \\
u^1_{maxd} & \cdots & u^1_{maxd+ndft-1} \\
u^1_{maxd-1} & \cdots & u^1_{maxd-1+ndft-1} \\
\vdots & & \vdots \\
u^1_{maxd-m_1+1} & \cdots & u^1_{maxd-m_1+1+ndft-1} \\
u^2_{maxd} & \cdots & u^2_{maxd+ndft-1} \\
\vdots & & \vdots \\
\vdots & & \vdots \\
u^{n_{in}}_{maxd-m_{n_{in}}+1} & \cdots & u^{n_{in}}_{maxd-m_{n_{in}}+1+ndft-1}
\end{bmatrix} \tag{3.3}$$

Output set matrix:

$$\begin{bmatrix}
y^1_{maxd+1} & \cdots & y^1_{maxd+1+ndft-1} \\
y^2_{maxd+1} & \cdots & y^2_{maxd+1+ndft-1} \\
\vdots & & \vdots \\
\vdots & & \vdots \\
y^{n_{out}}_{maxd+1} & \cdots & y^{n_{out}}_{maxd+1+ndft-1}
\end{bmatrix} \tag{3.4}$$

Where

ns	Is the total number of samples.
ph	Is the number of future predictions added to the cost. $ph = 1$ for one step ahead prediction training.
$maxd = max(m_1, \cdots, m_{n_{in}}, n_1, \cdots, n_{n_{out}})$	Is the max. time delay for all inputs and feedback outputs.
$ndft = ns - maxd - ph$	Is the number of data points for training.
n_i	Is the number of delayed feed back outputs to the network for output i.
m_i	Is the number of delayed signals for input i.
y_j^i	Is output number i for time j.
u_j^i	Is input number i for time j.

Training a neural network with the `NetworkTrainer` class utilizing the input and output set matrices in equation 3.3 and 3.4 respectively will, if the training data was adequately exciting, result in a neural network capable of predicting the desired system output one step ahead of time as in equation 3.1.

A General Dynamic Neural Network Function

A general definition of a dynamic neural network output function capable of handling multiple inputs, outputs and data sets is necessary in order to assist further mathematical analysis.

$$Y_{NN} = NeuralNetwork(X_1, X_2, \cdots, X_n, Z_1, Z_2, \cdots, Z_m, W) \tag{3.5}$$

$$X_i = \begin{bmatrix} {}^1x_i^1 \\ {}^2x_i^1 \\ \vdots \\ {}^{ndft}x_i^1 \\ {}^1x_i^2 \\ \vdots \\ \vdots \\ {}^{ndft}x_i^{n_{out}} \end{bmatrix}, \; Z_i = \begin{bmatrix} {}^1z_i^1 \\ {}^2z_i^1 \\ \vdots \\ {}^{ndft}z_i^1 \\ {}^1z_i^2 \\ \vdots \\ \vdots \\ {}^{ndft}z_i^{n_{in}} \end{bmatrix}, \tag{3.6}$$

Where

$n = max(n_1, \cdots, n_{n_{out}})$	Is the maximum output time delay needed by the neural network to model the system network
$m = max(m_1, \cdots, m_{n_{in}})$	Is the maximum input time delay needed by the neural network to model the system.
$X_1 \cdots X_n$	Is the feed back system outputs for time k to $k - n + 1$ for all sets.
$Z_1 \cdots Z_m$	Is the system inputs for time k to $k - n + 1$ for all sets.
${}^i x_k^j$	Is the system output number j, time k, set i.
${}^i z_k^j$	Is the system input number j, time k, set i.
W	Is the neural network weight vector.

The dynamic neural network function shown in equation 3.5 is the functional representation of the graph in figure 3.1.

Neural Network Model Training Problems

Neural network one step ahead prediction for one data set utilizing the inputs and outputs, U_k and Y_k in equation 3.2, is calculated with a *trained* network in the following way.

$$\hat{Y}_{k+1} = NeuralNetwork(Y_k, Y_{k-1}, \cdots, Y_{k-n+1}, U_k, U_{k-1}, \cdots, U_{k-m+1}, W_T) \tag{3.7}$$

Where

W_T	Is the trained neural network weights.

This training of such a dynamic neural network can however go wrong in several ways.

1. The neural network model converges towards the naive predictor.
2. The neural network model becomes unstable.
3. The neural network model output diverges in some areas.

The Naive Predictor Case (1)

The neural network could becomes a naive predictor:

$$\begin{aligned}\hat{Y}_{k+1} &= NeuralNetwork(Y_k, Y_{k-1}, \cdots, Y_{k-n+1}, U_k, U_{k-1}, \cdots, U_{k-m+1}, W_T) \\ &= Y_k\end{aligned} \tag{3.8}$$

This happens if the cost function (See equation 1.3) can obtain a small value when the neural networks acts as a naive predictor. This can happen if the system is for the most part relatively slow compared to the sampling time since the next value of the output will then be close the previous one.

The Neural Network Model Becomes Unstable (2)

The author of this work believes that the following two situations are the most frequently encountered in which the neural networks become unstable.

Unseen Unstable Region:

The neural network is usually trained on one data set and then tested on one or more other data set for verification.

The test data sets could make the network enter an operating point that it had not seen in the training set and this could be an unstable region for the neural network model. The network is simply stable for the operating points seen in the training set and not for all operating points in the test set.

Oscillating Training Data:

The training data could contain natural oscillations as in the case with the gasoline engines throttle air mass flow caused by pumping fluctuations.

The optimizing neural network training process could make the neural network converge towards a dynamic system with oscillating behavior since it would fit the training data.

The amplitude of the oscillations could however change with operating point since it is a nonlinear system and if the neural network enters an operating point it did not see during training then it might start oscillating with too large an amplitude

(Not really unstable, but...) or actually become unstable and go towards infinity.

The Neural Network Model Output Diverges in Some Areas (3)

Divergence in some areas of the neural network dynamic simulation can also occur if the neural network enters an operating point it did not see during training. The neural network then simply generates the wrong prediction and starts to move in the wrong direction.

It is also possible that the network simply "fell" into a local minimum during the training (optimization) process where certain areas are not predicted correctly. The error surface of the cost function is extremely complicated and contains many local minima.

3.2 Predictive Neural Network Model Training

All three cases of neural network modelling failure described in the previous section have the fact in common that unseen dynamics during the training can make the neural network malfunction.

A great part of the problem lies in the fact that the standard neural network training process is only intended for one step ahead predictions. The cost function (See equation 1.3) only adds one step ahead prediction errors (Squared) to the total cost.

All three problems mentioned in the previous section can be helped by adding 1 to ph (Short for prediction horizon) step ahead prediction errors to the cost function as shown in equation 3.9. However, the larger ph is the longer the training time will be and the larger the memory consumption will be too.

The choice of ph should therefore be as small as possible, but large enough to help solve the previously mentioned dynamic neural network training problems.

The predictive nature of the following cost function now requires that the data sets are given to the algorithm in the same order as they were sampled in. The indices in equation 3.9 refer to the data set number which is now also the sample time. An index of 1 is the first sample and a set index of 2 the next sample and so on.

$$J_{predictive} = \sum_{i=maxd+1}^{maxd+1+ndft} \left(\sum_{j=1}^{ph} \left(R_{i+j-1} - \hat{Y}_{i,j} \right)^T \left(R_{i+j-1} - \hat{Y}_{i,j} \right) \right) \tag{3.9}$$

$$\begin{aligned}
\hat{Y}_{i,1} &= NeuralNetwork(Y_{i-1}, Y_{i-2}, \cdots, Y_{i-n}, U_{i-1}, \cdots, U_{i-m}, W) \\
\hat{Y}_{i,2} &= NeuralNetwork(\hat{Y}_{i-1,1}, Y_{i-2}, \cdots, Y_{i-n}, U_{i-1}, \cdots, U_{i-m}, W) \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\hat{Y}_{i,j} &= NeuralNetwork(\hat{Y}_{i-1,j-1}, \hat{Y}_{i-2,j-2}, \cdots, \hat{Y}_{i-n,j-n}, U_{i-1}, \cdots, U_{i-m}, W)
\end{aligned} \tag{3.10}$$

Where

R_i	Is the desired neural network outputs for set i.
$\hat{Y}_{i,j}$	Is the j step ahead neural network predicted outputs for output set i, based on $j-1$ previous predictions.

The neural network weight size term (Regularization or weight decay, see section 1.3.2) in the cost function in equation 1.3 is left out in the description of the predictive algorithm since it is relatively simple to implement and is not of interest in the following sections.

This predictive training strategy will reduce the probability of encountering the problems mentioned previously because.

1. The naive predictor will most likely cause the cost function to attain a higher value than before since the signal now has a larger number of samples in which it can change and thus make a naive prediction more wrong.

2. An unstable network will make the output diverge quickly towards infinity and over a larger number of predictions cause the cost function to attain a much higher value than a stable network would.

3. It is more unlikely that the network will produce the wrong predictions because it fell into a "bad" minimum since that would also cause the cost function to attain a much higher value than a minimum where all the prediction are close to the target outputs.

3.3 Predictive Neural Network Training Algorithm

The neural network system model training problems described in section 3.1.2 is frequently encountered when training engine system models and the predictive training described in section 3.2 is thus very helpful.

The mathematical components for a predictive neural network training algorithm will be developed in this section and an overview of the entire training algorithm

is described.

Neural network training requires the utilization of an optimization algorithm that minimizes a cost function as explained in section 1.3.2. The optimization algorithm chosen for this work is the Levenberg-Marquardt algorithm because of its robustness, simplicity and efficiency. See for instance [19].

The mathematics necessary to program an efficient predictive neural network training class (with respect to training time) is the calculation of the predictive cost function in equation 3.9 in a way that is suitable for the Levenberg-Marquardt algorithm and an analytic derivative of the error function f_i in the Levenberg-Marquardt cost function in equation A.3 (To increase training speed).

The Levenberg-Marquardt optimization algorithm needs the first derivatives of the error functions f_i in the cost function in equation A.3 in order to calculate the optimizing step.

This is usually not a big problem with the standard squared errors cost function (See equation 1.3), but becomes somewhat more complicated when the cost function contains predictions based on predictions as in the predictive cost function in equation 3.9.

The complexity is only intensified by the desired to make the algorithm able to handle all given training data simultaneously for a multi input multi output network which is the most efficient and practical method with respect to training time.

3.3.1 The Predictive Cost Function

The neural network library developed in C++ for this work is described in appendix A.7 and equation A.5 and A.6 illustrates how the arbitrary number of inputs and outputs are handled for any number of input-output sets. It is a matter of data organization.

It is important to organize the data in such a way that they can be written in vector form for application of multi dimensional calculus using the chain rule and the product differentiation rule mainly.

The inputs and outputs with respect to their time delays are organized as in equation 3.3 and 3.4.

The Levenberg-Marquardt optimization algorithm is designed to handle a cost function that looks like this.

$$\begin{aligned} J_{marq} &= (Y-R)^T(Y-R) \\ &= E^T E \end{aligned} \tag{3.11}$$

which means that all outputs for all data sets must be packed into a vector in order to match the Levenberg-Marquardt algorithms cost function format.

The cost function in equation 3.9 is not in the correct format but if the vectors R and Y in equation 3.11 are formatted in the correct way then 3.11 can be made equal to 3.9.

The following formats for R and Y will make 3.11 equal to 3.9.

$$R = \begin{bmatrix} r^1_{maxd+1} \\ \vdots \\ r^1_{maxd+1+ndft-1} \\ r^2_{maxd+1} \\ \vdots \\ \vdots \\ r^{n_{out}}_{maxd+1+ndft-1} \\ r^1_{maxd+2} \\ \vdots \\ \vdots \\ r^{n_{out}}_{maxd+2+ndft-1} \\ r^1_{maxd+3} \\ \vdots \\ \vdots \\ \vdots \\ r^{n_{out}}_{maxd+ph+ndft-1} \end{bmatrix}, \; Y = \begin{bmatrix} \hat{y}^1_{maxd+1,1} \\ \vdots \\ \hat{y}^1_{maxd+1+ndft-1,1} \\ \hat{y}^2_{maxd+1,1} \\ \vdots \\ \vdots \\ \hat{y}^{n_{out}}_{maxd+1+ndft-1,1} \\ \hat{y}^1_{maxd+2,2} \\ \vdots \\ \vdots \\ \hat{y}^{n_{out}}_{maxd+2+ndft-1,2} \\ \hat{y}^1_{maxd+3,3} \\ \vdots \\ \vdots \\ \vdots \\ \hat{y}^{n_{out}}_{maxd+ph+ndft-1,ph} \end{bmatrix} \tag{3.12}$$

Where

r^j_i	Is the target output j for set i.
$\hat{y}^j_i, k$	Is the k step ahead predicted output j for set i.

The derivative needed by the Levenberg-Marquardt optimization algorithm is.

$$\frac{dE}{dW} \tag{3.13}$$

Where W is a vector containing all the neural network weights and those are the parameters to be found in the optimization process in order to make the neural network produce the target outputs R when it is given the inputs U.

The weight vector W has to have a chosen format which is kept standard for all applications and the chosen format can be found in the neural network library appendix A.7 in equation A.11.

3.3.2 The predictions - A Closer Look - Derivatives

The prediction error derivative in equation 3.13 is equal to.

$$\frac{dE}{dW} = \frac{dY}{dW} \tag{3.14}$$

and Y in equation 3.12 contains predictions based on previous predictions which makes things complicated.

However the chain rule makes it possible calculate the derivative of the the predictions in an iterative way that utilizes the derivative for time k to calculate the derivative for time $k+1$. This will be shown in the following.

It is necessary to write up the predictions in equation 3.10 in a more compact form including the neural network weight vector W as a parameter to make the derivation clearer. This is shown in equation 3.16.

The prediction equations will now also represent directly how the neural network library in appendix A.7 works in the sense that the neural network output function, here NN, is capable of taking multiple input sets and output the corresponding multiple output sets.

The input and output vectors in equation 3.2 are thus redefined to include all data sets as follows.

$$Y_{k+i} = \begin{bmatrix} y^1_{maxd+i} \\ y^1_{maxd+i+1} \\ \vdots \\ y^1_{maxd+i+ndft} \\ y^2_{maxd+i} \\ \vdots \\ \vdots \\ y^{n_{out}}_{maxd+i+ndft} \end{bmatrix}, U_{k+i} = \begin{bmatrix} u^1_{maxd+i} \\ u^1_{maxd+i+1} \\ \vdots \\ u^1_{maxd+i+ndft} \\ u^2_{maxd+i} \\ \vdots \\ \vdots \\ u^{n_{in}}_{maxd+i+ndft} \end{bmatrix}, i \leq ph \tag{3.15}$$

$$\begin{aligned} \hat{Y}_{k+1} &= NN(Y_k, Y_{k-1}, \cdots, Y_{k-n+1}, U_k, \cdots, U_{k-m+1}, W) &&= NN(1) \\ \hat{Y}_{k+2} &= NN(\hat{Y}_{k+1}, Y_k, \cdots, Y_{k-n+2}, U_{k+1}, \cdots, U_{k-m+2}, W) &&= NN(2) \\ &\vdots &&\vdots \\ \hat{Y}_{k+j} &= NN(\hat{Y}_{k+j-1}, \hat{Y}_{k+j-2}, \cdots, \hat{Y}_{k+j-n}, U_{k+j-1}, \cdots, U_{k+j-m}, W) &&= NN(j) \end{aligned} \tag{3.16}$$

Where

NN(i) Is the neural network output predicting $\hat{Y}_{k+i}$ with the necessary input arguments to do so.

The $\hat{}$ just indicates that this is a prediction and not a measurement. The data format is however the same as in 3.15. Furthermore the prediction equations 3.16 are now in the same format as the general definition in equation 3.5 which is more convenient in the following.

The predictions in equation 3.16 are the elements in the vector Y in equation 3.12. The vector $\hat{Y}_{k+1}$ is the first $n_{out} \times ndft$ elements in the vector Y and the vector $\hat{Y}_{k+2}$ is the next $n_{out} \times ndft$ elements and so on.

In order to calculate the derivative 3.14 it is thus necessary to find the derivatives of the vectors $\hat{Y}_{k+1} \cdots \hat{Y}_{k+ph}$ with respect to the neural network weight vector W.

The derivative for the first prediction in equation 3.16 is simple since there are no previous predictions in the neural networks inputs.

$$\frac{d\hat{Y}_{k+1}}{dW} = \frac{dNN}{dW}(1) \tag{3.17}$$

The chain rule is applied for the remaining predictions for each input to NN that is a prediction itself, but no more than there are feedback outputs present in the

input argument list to NN which is a maximum of n times.

$$\begin{array}{rcl} \frac{d\hat{Y}_{k+2}}{dW} & = & NN_1^{'}(2)\frac{d\hat{Y}_{k+1}}{dW} + \frac{dNN}{dW}(2) \\ \frac{d\hat{Y}_{k+3}}{dW} & = & NN_1^{'}(3)\frac{d\hat{Y}_{k+2}}{dW} + NN_2^{'}(3)\frac{d\hat{Y}_{k+1}}{dW} + \frac{dNN}{dW}(3) \\ \vdots & & \vdots \\ \frac{d\hat{Y}_{k+ph}}{dW} & = & \sum_{i=1}^{min(n,ph-1)} \left(NN_i^{'}(ph)\frac{d\hat{Y}_{k+ph-i}}{dW} \right) + \frac{dNN}{dW}(ph) \end{array} \tag{3.18}$$

Where

$NN_i^{'}(j)$ Is the partial derivative of the neural network output function given the inputs to predict $\hat{Y}_{k+j}$ as in 3.16 with respect to input X_i.

3.3.3 Dynamic Neural Network Partial Derivatives

The partial derivatives with respect to an input vector X_i will, utilizing the general dynamic neural network output function format in equation 3.5 look as follows.

$$NN_i^{'}(j) = \begin{bmatrix} \frac{d^1y^1_{NN}}{d^1x^1_i} & \frac{d^1y^1_{NN}}{d^2x^1_i} & \cdots & \frac{d^1y^1_{NN}}{d^{ndft}x^1_i} & \frac{d^1y^1_{NN}}{d^1x^2_i} & \cdots\cdots & \frac{d^1y^1_{NN}}{d^{ndft}x^{nout}_i} \\ \frac{d^2y^1_{NN}}{d^1x^1_i} & \frac{d^2y^1_{NN}}{d^2x^1_i} & \cdots & \frac{d^2y^1_{NN}}{d^{ndft}x^1_i} & \frac{d^2y^1_{NN}}{d^1x^2_i} & \cdots\cdots & \frac{d^2y^1_{NN}}{d^{ndft}x^{nout}_i} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{d^{ndft}y^1_{NN}}{d^1x^1_i} & \frac{d^{ndft}y^1_{NN}}{d^2x^1_i} & \cdots & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^1_i} & \frac{d^{ndft}y^1_{NN}}{d^1x^2_i} & \cdots\cdots & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^{nout}_i} \\ \frac{d^1y^2_{NN}}{d^1x^1_i} & \frac{d^1y^2_{NN}}{d^2x^1_i} & \cdots & \frac{d^1y^2_{NN}}{d^{ndft}x^1_i} & \frac{d^1y^2_{NN}}{d^1x^2_i} & \cdots\cdots & \frac{d^1y^2_{NN}}{d^{ndft}x^{nout}_i} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{d^{ndft}y^{nout}_{NN}}{d^1x^1_i} & \frac{d^{ndft}y^{nout}_{NN}}{d^2x^1_i} & \cdots & \frac{d^{ndft}y^{nout}_{NN}}{d^{ndft}x^1_1} & \frac{d^{ndft}y^{nout}_{NN}}{d^1x^2_i} & \cdots\cdots & \frac{d^{ndft}y^{nout}_{NN}}{d^{ndft}x^{nout}_i} \end{bmatrix} \tag{3.19}$$

The indices of the elements of the matrix in equation 3.19 has the following meaning.

$$\frac{d^{\text{output set}}y^{\text{output nr.}}_{NN}}{d^{\text{input set}}x^{\text{feed back output nr.}}_{\text{argument nr. in NN}}} \tag{3.20}$$

Most of the elements in the matrix in equation 3.19 is however zero since each neural network output data set only depends on one input data set.

The elements where the *input set* and *output set* indices are different are thus zero.

$$\frac{d^{\textit{output set}}y_{NN}^{\textit{output nr.}}}{d^{\textit{input set}}x_{\textit{argument nr. in NN}}^{\textit{feed back output nr.}}} = 0\ ,\ \textit{input set} \neq \textit{output set} \tag{3.21}$$

Furthermore, the elements for which $n_{\textit{feed back output nr.}} < \textit{argument nr. in NN}$ are also zero since the time delayed feed back output represented by the signal

$$^{\textit{input set}}x_{\textit{argument nr. in NN}}^{\textit{feed back output nr.}} \tag{3.22}$$

is then not utilized when calculating the neural network output.

$$\frac{d^{\textit{output set}}y_{NN}^{\textit{output nr.}}}{d^{\textit{input set}}x_{\textit{argument nr. in NN}}^{\textit{feed back output nr.}}} = 0\ ,\ if\ n_{\textit{feed back output nr.}} < \textit{argument nr. in NN} \tag{3.23}$$

It may look like the feed back output with $n_{\textit{feed back output nr.}} < \textit{argument nr. in NN}$ is utilized in the general neural network output function format in equation 3.16 and 3.5 since the input argument list includes feed back outputs down to $k-n+1$ (in 3.16) which is the maximum time delay for all feed back outputs.

Equation 3.5 is however only a general mathematical format practical for calculating the derivatives and the programmed version of the neural network does not take inputs that are not used in the calculation of the outputs since that would be inefficient with respect to execution speed. The programmed version of the neural network output is calculated as shown in figure 3.1.

The matrix 3.19 is thus best treated as a sparse matrix since most of the elements are zero for large $ndft$.

Treating the matrix 3.19 as sparse also solves the problem of having to create and store this huge matrix. The non sparse version of the matrix consist of $no\ ndft \times no\ ndft$ elements. The output data sets for a single output system sampled for 20 seconds at 200 Hz produces 4000 samples. The partial derivative matrix in equation 3.19 will then, not taking $maxd$ and ph into consideration since they are usually small compared to the number of training data sets, take up about.

$$\frac{4000 \times 4000 \times 8}{1024 \times 1024} \approx 122.07MB \tag{3.24}$$

Which is too large an amount of memory for one matrix if the execution speed should be kept reasonably small on a standard computer.

Sparse Matrix Product

It is much more efficient with respect to execution time and memory consumption to construct an algorithm that performs the matrix product.

$$NN_i^{'}(j)\frac{d\hat{Y}_{k+j-i}}{dW} \tag{3.25}$$

without constructing the full matrix in equation 3.19.

The non zero elements in the neural network partial derivative in equation 3.19 are the elements from the derivative of the neural network output function with respect to the neural network inputs shown in equation 3.26.

$$\frac{dY_{NN}}{dX} = \begin{bmatrix} \frac{d^1y^1_{NN}}{d^1x^1_1} & \frac{d^1y^1_{NN}}{d^1x^1_2} & \cdots & \frac{d^1y^1_{NN}}{d^1x^1_{n_1}} & \frac{d^1y^1_{NN}}{d^1x^2_1} & \cdots\cdots & \frac{d^1y^1_{NN}}{d^1x^2_{n_{in}}} \\ \frac{d^2y^1_{NN}}{d^2x^1_1} & \frac{d^2y^1_{NN}}{d^2x^1_2} & \cdots & \frac{d^2y^1_{NN}}{d^2x^1_{n_1}} & \frac{d^2y^1_{NN}}{d^2x^2_1} & \cdots\cdots & \frac{d^2y^1_{NN}}{d^2x^2_{n_{in}}} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^1_1} & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^1_2} & \cdots & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^1_{n_1}} & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^2_1} & \cdots\cdots & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^2_{n_{in}}} \\ \frac{d^1y^2_{NN}}{d^1x^1_1} & \frac{d^1y^2_{NN}}{d^1x^1_2} & \cdots & \frac{d^1y^2_{NN}}{d^1x^1_{n_1}} & \frac{d^1y^2_{NN}}{d^1x^2_1} & \cdots\cdots & \frac{d^1y^2_{NN}}{d^1x^2_{n_{in}}} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^1_1} & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^1_2} & \cdots & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^1_{n_1}} & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^2_1} & \cdots\cdots & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^2_{n_{in}}} \end{bmatrix} \tag{3.26}$$

The `SHLNetwork` class in the neural network library table already implements a method to calculate this matrix (See appendix A.7 table A.14).

A closer look at the partial derivative matrix in equation 3.19 utilizing the conditions in equation 3.21 and 3.23 reveals how the elements from the neural network input derivative in equation 3.26 are placed in the neural network partial derivative matrix 3.19.

The neural network partial derivative 3.19 with the appropriate elements set to zero is shown in equation 3.27. It is however assumed in 3.27 that condition 3.23 is never active.

$$NN_i^{'}(j) =$$

$$\begin{array}{c|c|c}
\overbrace{\qquad\qquad}^{\textbf{f.b.g. 1}\ ndft} & \overbrace{\qquad\qquad}^{\textbf{f.b.g. 2}\ ndft} & \overbrace{\qquad}^{\textbf{f.b.g. 3-n}_{\textbf{out}}\ (n_{out}-2)ndft} \\
\left[\begin{array}{cccc}
\frac{d^1y^1_{NN}}{d^1x^1_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^1_{NN}}{d^2x^1_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^1_i} \\
\frac{d^1y^2_{NN}}{d^1x^1_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^2_{NN}}{d^2x^1_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^2_{NN}}{d^{ndft}x^1_i} \\
\vdots & & & \vdots \\
\vdots & & & \vdots \\
\frac{d^1y^{n_{out}}_{NN}}{d^1x^1_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^{n_{out}}_{NN}}{d^2x^1_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^1_i}
\end{array}\right. &
\left.\begin{array}{cccc}
\frac{d^1y^1_{NN}}{d^1x^2_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^1_{NN}}{d^2x^2_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^1_{NN}}{d^{ndft}x^2_i} \\
\frac{d^1y^2_{NN}}{d^1x^2_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^2_{NN}}{d^2x^2_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^2_{NN}}{d^{ndft}x^2_i} \\
\vdots & & & \vdots \\
\vdots & & & \vdots \\
\frac{d^1y^{n_{out}}_{NN}}{d^1x^2_i} & 0 & \cdots & 0 \\
0 & \frac{d^2y^{n_{out}}_{NN}}{d^2x^2_i} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \frac{d^{ndft}y^{n_{out}}_{NN}}{d^{ndft}x^2_i}
\end{array}\right. &
\left.\begin{array}{c}
\cdots\cdots\cdots \\
\\
\\
\cdots\cdots\cdots \\
\cdots\cdots\cdots \\
\\
\\
\cdots\cdots\cdots \\
\\
\\
\cdots\cdots\cdots \\
\\
\\
\cdots\cdots\cdots
\end{array}\right]
\end{array} \tag{3.27}$$

$f.b.g.$ refers to Feed Back Group and separates the columns into groups each belonging to a feed back output.

The condition in equation 3.23 will make all the elements in feed back group *feed back output nr.* zero if $n_{\textit{feed back output nr.}} <$ *argument nr. in NN* for that feed back output.

The non zero elements in the neural network partial derivative 3.19

$$NN_i^{'}$$

are the elements from the n_{out} columns

$$i, i + n_1, i + n_2, \cdots, i + n_{n_{out}-1} \tag{3.28}$$

in the neural network input derivative in equation 3.26 if the input structure from figure 3.1 and the dynamic input set matrix in equation 3.3 is applied since the feed back outputs and their respective time delayed versions are then in the first

$$S_n = \sum_{i=1}^{nout} n_i \tag{3.29}$$

S_n columns of the neural network input derivative matrix in equation 3.26.

The sparse matrix product in equation 3.25 can then be constructed in a double loop picking out the elements from the columns of the neural network input derivative in equation 3.26 specified in equation 3.27 and 3.28 for each row in 3.26 and utilize them as the coefficients for a linear combination of the rows in

$$\frac{d\hat{Y}_{k+j-i}}{dW} \tag{3.30}$$

corresponding to their position in the neural network partial derivative matrix 3.19. The rows of the sparse matrix product are those linear combinations.

A C++ class implementation of the regular one step ahead training and the predictive training algorithm has been developed and the usage is described in appendix A.7.5. The source code can be found in the Matrix Control Library/NetworkTrainer folder on the source code appendix CD coming with this dissertation.

3.3.4 Demonstration of the Predictive Training Algorithm

The training difficulties mentioned in section 3.1.2 are severely reduced when utilizing the new predictive training algorithm. An example will show just how big the effect of taking more than one prediction into account in the cost function.

The example is the training of a neural network to model the throttle air mass flow in a British Leyland 1.275 L 4 cylinder engine. The example neural network configuration is as shown in table 3.3.4.

Figure 3.2 shows how well the network training was able to optimize the neural network. The training error is very small except for in a very few places.

Example Neural Network Configuration			
	Input	**Unit**	**Time**
Inputs:	Throttle Plate Angle Command	[°]	k
	Throttle Plate Angle Command	[°]	k-1
	Pressure Ratio $\left(\frac{Pman}{Pamb}\right)$	[]	k
	Throttle Air Mass Flow (Feed Back)	$\left[\frac{g}{s}\right]$	k
Output:	Throttle Air Mass Flow	$\left[\frac{g}{s}\right]$	k+1
#Neurons	5	[]	

Table 3.1: Throttle Air Mass Flow Example Neural Network Configuration

Figure 3.3 shows how the neural network performs when utilized as a dynamic model. The network quickly and often diverges from the corresponding measurements and is not at all a good model of the throttle air mass flow.

The errors seen in figure 3.3 will be much less likely to happen with predictive neural network training since the cost function would grow very large very quickly this way and this will produce a larger gradient in the correct direction in weight space.

Figure 3.4 shows how a neural network, utilizing the same configuration as in table 3.3.4 and trained with the predictive training algorithm on the same training data as those for the one step ahead neural network, performs when utilized as a dynamic model of the throttle air mass flow. A prediction horizon (ph) of 5 was utilized to train the neural network with the predictive training algorithm.

The improvement is immediately obvious. Results like those for the one step ahead training in figure 3.3 are common and the predictive training algorithm significantly reduces the number of such "training accidents".

3.4 Levenberg Marquardt Algorithms Literature

3.4.1 Background

The literature has for some time now put a lot of effort into improving various training algorithms. The back propagation (BP) algorithm has especially received a lot of attention and many modifications has been made to the BP algorithm (BP with momentum [1], BP with adaptive learning rate [18], BP with conjugate gradients [11], BP with Levenberg Marquardt [14] and many others).

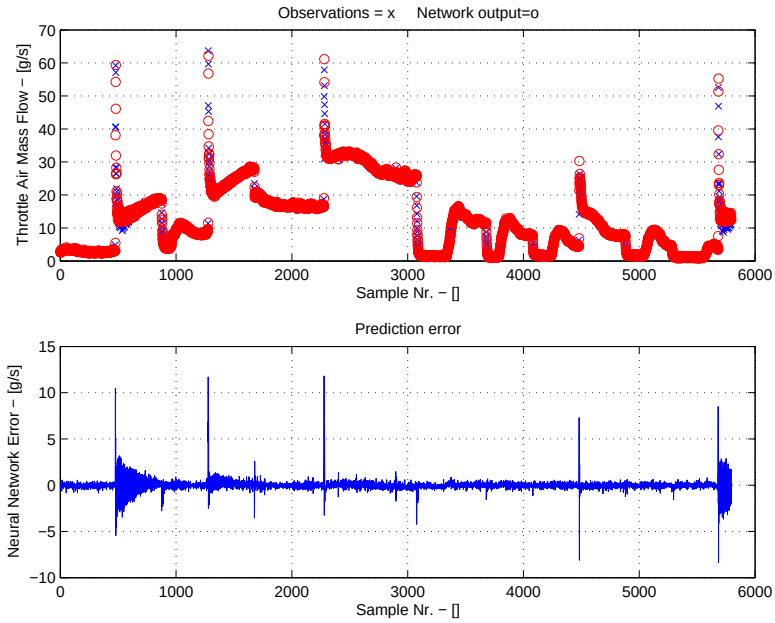


Figure 3.2: One Step Ahead Training Results

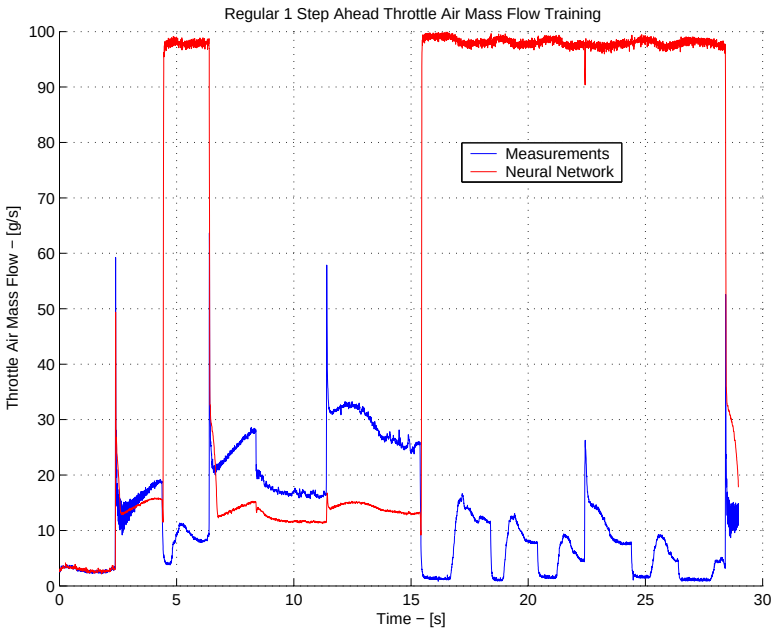


Figure 3.3: One Step Ahead Training - Simulation Example

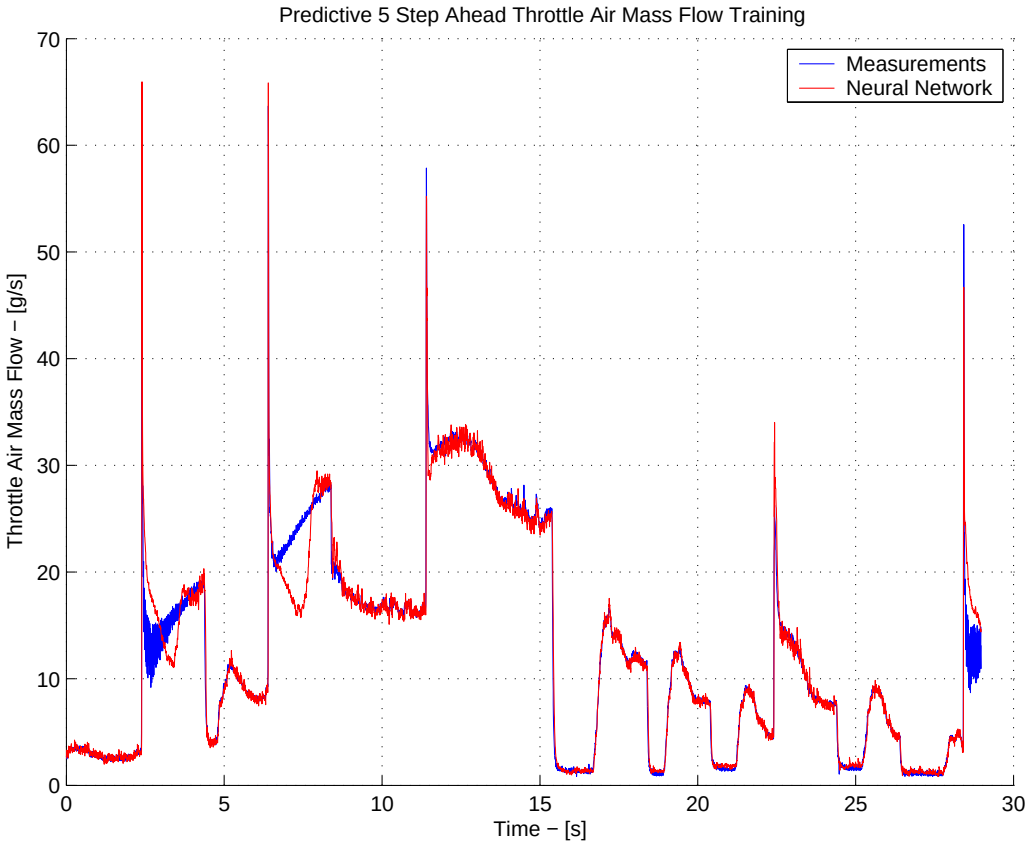


Figure 3.4: Five Step Ahead Predictive Training - Simulation Example

All these BP methods focus on improving the neural network training on several aspects.

- Fast training time.
- Ability to avoid local minima.
- High success Rate.
- High accuracy.

All these many modifications to the BP algorithm have turned it into a method for dealing with multi layered neural networks rather than a first order steepest descent algorithm as it was at first.

It is interesting to note that a method such as the BP with momentum algorithm actually utilizes second order information by adding the momentum term to the weight update rule. This is a general trend in the literature. The use of second order information in the update rules for neural network weights has turned out to be a successful strategy.

The Levenberg Marquardt (LM) algorithm has especially turned out to be popular because of its fast convergence abilities. Many papers (i.e. [14], [4], [5], [45], [23], [13] and [43]) utilize the Levenberg-Marquardt algorithm in some form.

The paper [14] describes how to implement the BP algorithm with an LM weight update. This paper is an example of how the BP is more an algorithm for dealing with multi layered neural networks and how second order methods such as LM (and particularly LM based methods) are efficient neural network training algorithms.

This choice of utilizing the LM algorithm in this dissertation was natural since the literature and personal experience of the author has shown that LM is one of the best and simplest training algorithms available.

3.4.2 Comparison of LM algorithms in the Literature with the Predictive Training Algorithm

The predictive training algorithm developed in this chapter works with single hidden layer neural networks, to keep things simple, and the BP feature is thus not

needed here. It will therefore not be discussed further.

Other Implementations of the LM Algorithm in the Literature

Some papers in the literature work on improving the LM update by modifying it in various ways.

The paper [4] is utilizing a different cost function with lagrange multipliers in order to ensure conjugate weight updates and a specified size for the trust region around the current point in weight space. This makes the training algorithm extremely robust and it succeeds where many other advances second order methods fail.

The paper [5] is modifying the cost function by "collapsing" chosen errors from the error vector into the square root of a sum of the chosen squared errors.

$$\hat{e}_i(w) = \sqrt{\sum_{a \in I_i} e_a^2(w)} \tag{3.31}$$

Where $\hat{e}(w)$ is the "collapsed" error referred to as the aggregated error in [5].

This makes it possible to reduce the size of the hessian matrix in the LM algorithm and thus reduce the time and memory needed for calculating the inverse hessian matrix.

The paper [45] is utilizing the LM algorithm in a recursive setup. The weights are update in real time sample by sample. The new aspect in this paper is that the authors show how to modify the recursive LM algorithm so that the weights can be update one by one in a parallel manner and thus improve computational efficiency.

The paper [23] is reducing the computational time by only working on a randomly chosen neighborhood of weights at each iteration.

The Utilization of the LM Algorithm in the Predictive Training Algorithm

The aim of this utilization of the LM algorithm in the predictive training algorithm is to improve all the items listed in the beginning of this section and particularly the points mentioned in section 3.1.2, 3.1.2, 3.1.2 in this chapter.

The training time for one batch was however increased by this method due to the enlarged jacobian matrix, but the success rate and quality of the dynamic neural network models was increased significantly and thus reducing the overall training time spent on producing a dynamic neural network model.

The LM algorithm is utilized in its basic form (see the description in section 5.1.4. The cost function is however changed to include neural network model predictions in order to produce a better gradient with respect to n step ahead prediction. This is similar to what is done in the paper [5] with the aggregated errors.

In the future it would be interesting and useful to incorporate the cost function elements utilizing lagrange multipliers as in [4] to increase the robustness of the predictive training algorithm.

It would also be very interesting to implement the aggregated error scheme from [5] in order to reduce the training time and memory consumption.

3.5 Neural Network Derivatives

Neural network training requires the utilization of the derivative of the neural network output function with respect to the weights. Furthermore, the predictive control algorithm described in chapter 5 also requires the derivative of the neural network output function with respect to the inputs of the neural network.

The intention with this section is to describe the derivation and programming of analytic expressions for the derivatives of the neural network output function. This requires that a specific neural network is chosen.

The choice in this work falls upon the single hidden layer neural network since it is a widely utilized type of neural network (See section 1.3) and it is a mathematically relative simple network structure that still has great system modelling capabilities.

3.5.1 The Neural Network Output Function

The single hidden layer neural network output function looks like this in matrix form.

$$Y_{nn} = W_2 ActivationFunction(W_1 X + B_1) + B_2 \tag{3.32}$$

Where

W_1 is the hidden layer weight matrix.

$$W_1 = \begin{bmatrix} w_{111} & \cdots & w_{11n_{in}} \\ \vdots & \ddots & \vdots \\ w_{1n_h1} & \cdots & w_{1n_hn_{in}} \end{bmatrix} \tag{3.33}$$

B_1 is the hidden layer biases.

$$B_1 = \begin{bmatrix} b_{11} \\ \vdots \\ b_{1n_h} \end{bmatrix} \tag{3.34}$$

W_2 is the output layer weights.

$$W_2 = \begin{bmatrix} w_{211} & \cdots & w_{21n_{in}} \\ \vdots & \ddots & \vdots \\ w_{2n_{out}1} & \cdots & w_{2n_{out}n_{in}} \end{bmatrix} \tag{3.35}$$

B_2 is the output layer biases.

$$B_2 = \begin{bmatrix} b_{21} \\ \vdots \\ b_{2n_{out}} \end{bmatrix} \tag{3.36}$$

X is the neural network input vector.

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_{n_{in}} \end{bmatrix} \tag{3.37}$$

n_{in}	Is the number of inputs of the neural network.
n_h	Is the number of neurons in the neural network.
n_{out}	Is the number of outputs of the neural network.

The $ActivationFunction()$ function returns a vector where each element is the scalar version of $ActivationFunction()$, which is called $activationfunction()$ with lower case letter, of the corresponding element in the input vector to $ActivationFunction()$. The activation function of a neuron is described in section 1.3.

$$ActivationFunction\left(\begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}\right) = \begin{bmatrix} activationfunction(z_1) \\ \vdots \\ activationfunction(z_n) \end{bmatrix} \tag{3.38}$$

Define

$$\begin{aligned} T_i = T(I_i) =& activationfunction(I_i) \\ =& activationfunction(w_{1i1}x_1 + w_{1i2}x_2 + \cdots + w_{w1in_{in}}x_{n_{in}}) \end{aligned} \tag{3.39}$$

The outputs of the neural network can then be written as.

$$Y_{nn} = \begin{bmatrix} y_1 \\ \vdots \\ y_{n_{out}} \end{bmatrix} = \begin{bmatrix} w_{211}T_1 + w_{212}T_2 + \cdots + w_{21n_h}T_{n_h} + B_{21} \\ \vdots \\ w_{2n_{out}1}T_1 + w_{2n_{out}2}T_2 + \cdots + w_{2n_{out}n_h}T_{n_h} + B_{2n_{out}} \end{bmatrix} \tag{3.40}$$

3.5.2 The Derivative with Respect to the Weights

$$\frac{dY_{nn}}{dW} \tag{3.41}$$

This is the necessary derivative to perform efficient on-line and off-line training of single hidden layer neural networks. The derivatives can be found with respect to the weights by utilizing equation 3.39 and 3.40.

The partial derivatives in the derivative in equation 3.41 can be written in a general form as follows.

Hidden weights:

$$\frac{\partial y_i}{\partial w_{1jk}} = w_{2ij}\dot{T}_i x_k \tag{3.42}$$
$$1 \geq i \leq n_{out}\,,\, 1 \geq j \leq n_h\,,\, 1 \geq k \leq n_{in}$$

Hidden biases:

$$\frac{\partial y_i}{\partial b_{1j}} = w_{2ij}\dot{T}_j \tag{3.43}$$
$$1 \geq i \leq n_{out}\,, 1 \geq j \leq n_h$$

Output weights:

$$\frac{\partial y_i}{\partial w_{2jk}} = \begin{cases} T_k & ,i = j \\ 0 & ,i \neq j \end{cases} \tag{3.44}$$
$$1 \geq i \leq n_{out}\,, 1 \geq j \leq n_{out}\,, 1 \geq k \leq n_h$$

Output biases:

$$\frac{\partial y_i}{\partial b_{2j}} = \begin{cases} 1 & ,i = j \\ 0 & ,i \neq j \end{cases} \tag{3.45}$$
$$1 \geq i \leq n_{out}\,, 1 \geq j \leq n_{out}$$

Where $\dot{T}_i$ is the derivative of the activation function with the input specified by i in 3.39.

3.5.3 The Derivative with Respect to the Inputs

$$\frac{dY_{nn}}{dX} \tag{3.46}$$

This is the necessary derivative to perform efficient minimization of a predictive control cost function in order to calculate the predictive control signal. The derivatives can be found with respect to the inputs by utilizing equation 3.39 and 3.40.

The partial derivatives in the derivative in equation 3.46 can be written in a general form as follows.

$$\frac{\partial y_i}{\partial x_j} = \sum_{k=1}^{n_h} w_{2ik}\dot{T}_k w_{1kj} \tag{3.47}$$

3.5.4 Multiple Input Sets

The derivatives have now been found for one input set, but have to be expanded for multiple input and output sets (See A.7.1 for an explanation about input and output sets).

This has to be done in order to be able to use them for training neural networks on all available data simultaneously. For both on-line and off-line training.

The neural network library works with the input and output set matrix format shown in equation A.5 and A.6 where the output sets are put in a column each. This is, however, not very practical when it comes to the derivatives since the derivatives are taken with respect to vectors.

This is because the Levenberg-Marquardt optimization algorithm in this work is designed to handle multiple input output sets by converting the function output matrices into vectors with the $ConvertToVec()$ function.

The $Convert2Vec()$ function is part of the matrix C++ library (See appendix A.5) which takes the transposed rows of a matrix and stacks them after each other to convert the matrix into a vector.

The neural network output function supporting multiple input output sets looks like this.

$$Y_{nn} = \begin{bmatrix} y_{11} & \cdots & y_{1n_s} \\ \vdots & & \vdots \\ y_{n_{out}1} & \cdots & y_{n_{out}n_s} \end{bmatrix} = \tag{3.48}$$

$$\begin{bmatrix} w_{211}T_{11} + w_{212}T_{21} + \cdots + w_{21n_h}T_{n_h1} + B_{21} & \cdots \\ \vdots & \\ w_{2n_{out}1}T_{11} + w_{2n_{out}2}T_{21} + \cdots + w_{2n_{out}n_h}T_{n_h1} + B_{2n_{out}} & \cdots \\ w_{211}T_{1n_s} + w_{212}T_{2n_s} + \cdots + w_{21n_h}T_{n_hn_s} + B_{21} & \\ \vdots & \\ w_{2n_{out}1}T_{1n_s} + w_{2n_{out}2}T_{2n_s} + \cdots + w_{2n_{out}n_h}T_{n_hn_s} + B_{2n_{out}} & \end{bmatrix}$$

Where

$$T_{ij} = T(I_{ij}) = activationfunction(w_{1i1}x_{1j} + w_{1i2}x_{2j} + \cdots + w_{w1in_{in}}x_{n_{in}j}) \tag{3.49}$$

and

n_s Is the number of input-output sets
x_{ij} Is neural network input i from input-output set j

The neural network input matrix looks like this.

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1n_s} \\ \vdots & & \vdots \\ x_{n_{in}1} & \cdots & x_{n_{in}n_s} \end{bmatrix} \tag{3.50}$$

3.5.5 Neural Network Derivative Matrix Formats

When taking the derivative of the neural network outputs with respect to the weights, a specific order of weights has to be chosen so that it is known which weight each column in the derivative corresponds to.

The following order will be used for the neural network weights.

$$\begin{aligned} W_{wv} = [&w_{111} \cdots w_{11n_{in}} \; w_{121} \cdots\cdots w_{1n_hn_{in}} \; b_{11} \cdots b_{1n_h} \\ &w_{211} \cdots w_{21n_h} \; w_{221} \cdots\cdots w_{2n_{out}n_h} \; b_{21} \cdots b_{2n_{out}}]^T \end{aligned} \tag{3.51}$$

The matrix format for the derivatives of the neural network with respect to the weights and for multiple input sets will be formatted in the following way.

$$\frac{\partial Y_{nn}}{\partial W_{wv}} = \begin{bmatrix} \frac{\partial y_{11}}{\partial w_{111}} & \cdots & \frac{\partial y_{11}}{\partial b_{2n_{out}}} \\ \frac{\partial y_{12}}{\partial w_{111}} & \cdots & \frac{\partial y_{12}}{\partial b_{2n_{out}}} \\ \vdots & & \vdots \\ \frac{\partial y_{1n_s}}{\partial w_{111}} & \cdots & \frac{\partial y_{1n_s}}{\partial b_{2n_{out}}} \\ \frac{\partial y_{21}}{\partial w_{111}} & \cdots & \frac{\partial y_{21}}{\partial b_{2n_{out}}} \\ \frac{\partial y_{22}}{\partial w_{111}} & \cdots & \frac{\partial y_{22}}{\partial b_{2n_{out}}} \\ \vdots & & \vdots \\ \frac{\partial y_{n_{out}n_s}}{\partial w_{111}} & \cdots & \frac{\partial y_{n_{out}n_s}}{\partial b_{2n_{out}}} \end{bmatrix} \tag{3.52}$$

The matrix format for the derivative of the neural network with respect to the inputs.

$$\frac{\partial Y_{nn}}{\partial X} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x_{1..n_{in}}} \\ \frac{\partial y_{12}}{\partial x_{1..n_{in}}} \\ \vdots \\ \frac{\partial y_{1n_s}}{\partial x_{1..n_{in}}} \\ \frac{\partial y_{21}}{\partial x_{1..n_{in}}} \\ \frac{\partial y_{22}}{\partial x_{1..n_{in}}} \\ \vdots \\ \frac{\partial y_{n_{out}n_s}}{\partial x_{1..n_{in}}} \end{bmatrix} \tag{3.53}$$

See also section A.7.1 for an explanation of how the neural network library works with this format and these derivatives.

3.5.6 Programming the Derivatives

The matrix library (See section A.5) contains many useful functions to construct the derivatives 3.52 and 3.53 in an easy and relatively efficient way. It could be made much more efficient by programming every little bit in assembler, but the matrix manipulation functions in the matrix library makes this kind of programming a lot easier while experimenting and still quite efficient. Ready code can then later on be programmed directly in assembler when the program structure has been found.

The neural network weight vector format structure is chosen as in 3.51 because it makes the calculation of the derivatives easier. Easier because it makes it possible to construct submatrices from the basic weight vector matrices (3.33,3.34,3.35 and 3.36) that can be utilized to construct the full derivative 3.52 by concatenation and inner products.

This is more easily seen if the partial derivatives 3.42, 3.43, 3.44 and 3.45 is inserted in the full derivative with respect to the weights 3.52 utilizing a compact notation.

3.5.7 Programming the Derivative with Respect to the Weights

It is better to split the the derivative $\frac{\partial Y_{nn}}{\partial W_{wv}}$ up into four parts each belonging to a group of weights. The hidden weights W_1, the hidden biases B_1, the output weight W_2 and the output biases B_2.

$$\frac{\partial Y_{nn}}{\partial W_{wv}} = \left[\begin{array}{c|c|c|c} YWTX & YWT & YT & YI \end{array} \right] \tag{3.54}$$

$YWTX$	Is the derivative of Y_{nn} with respect to the hidden weights W_1 in the order: $w_{111} \quad \cdots \quad w_{11n_{in}} \quad w_{121} \quad \cdots\cdots \quad w_{1n_hn_{in}}$
YWT	Is the derivative of Y_{nn} with respect to the hidden biases B_1 in the order $b_{11} \quad \cdots \quad b_{1n_h}$
YT	Is the derivative of Y_{nn} with respect to the output weights W_2 in the order: $w_{211} \quad \cdots \quad w_{21n_h} \quad w_{221} \quad \cdots\cdots \quad w_{2n_{out}n_h}$
YI	Is the derivative of Y_{nn} with respect to the output biases B_2 in the order $b_{21} \quad \cdots \quad b_{2n_h}$

$$
YWTX = \begin{bmatrix}
w_{211}\dot{T}_{11}x_{11} & \cdots & w_{211}\dot{T}_{11}x_{n_{in}1} & w_{212}\dot{T}_{21}x_{11} & \cdots & w_{212}\dot{T}_{21}x_{n_{in}1} & w_{213}\dot{T}_{31}x_{11} & \cdots\cdots & w_{21n_h}\dot{T}_{n_h1}x_{n_{in}1} \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
w_{211}\dot{T}_{1n_s}x_{1n_s} & \cdots & w_{211}\dot{T}_{1n_s}x_{n_{in}n_s} & w_{212}\dot{T}_{2n_s}x_{1n_s} & \cdots & w_{212}\dot{T}_{2n_s}x_{n_{in}n_s} & w_{213}\dot{T}_{3n_s}x_{1n_s} & \cdots\cdots & w_{21n_h}\dot{T}_{n_hn_s}x_{n_{in}n_s} \\
w_{221}\dot{T}_{11}x_{11} & \cdots & w_{221}\dot{T}_{11}x_{n_{in}1} & w_{222}\dot{T}_{21}x_{11} & \cdots & w_{222}\dot{T}_{21}x_{n_{in}1} & w_{223}\dot{T}_{31}x_{11} & \cdots\cdots & w_{22n_h}\dot{T}_{n_h1}x_{n_{in}1} \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
w_{221}\dot{T}_{1n_s}x_{1n_s} & \cdots & w_{221}\dot{T}_{1n_s}x_{n_{in}n_s} & w_{222}\dot{T}_{2n_s}x_{1n_s} & \cdots & w_{222}\dot{T}_{2n_s}x_{n_{in}n_s} & w_{223}\dot{T}_{3n_s}x_{1n_s} & \cdots\cdots & w_{22n_h}\dot{T}_{n_hn_s}x_{n_{in}n_s} \\
w_{231}\dot{T}_{11}x_{11} & \cdots & w_{231}\dot{T}_{11}x_{n_{in}1} & w_{232}\dot{T}_{21}x_{11} & \cdots & w_{232}\dot{T}_{21}x_{n_{in}1} & w_{233}\dot{T}_{31}x_{11} & \cdots\cdots & w_{23n_h}\dot{T}_{n_h1}x_{n_{in}1} \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
w_{2n_{out}1}\dot{T}_{1n_s}x_{1n_s}\cdots & w_{2n_{out}1}\dot{T}_{1n_s}x_{n_{in}n_s} & w_{2n_{out}2}\dot{T}_{2n_s}x_{1n_s}\cdots & w_{2n_{out}2}\dot{T}_{2n_s}x_{n_{in}n_s} & w_{2n_{out}3}\dot{T}_{3n_s}x_{1n_s}\cdots\cdots & w_{2n_{out}n_h}\dot{T}_{n_hn_s}x_{n_{in}n_s}
\end{bmatrix} \tag{3.55}
$$

$$YWT = \begin{bmatrix} w_{211}\dot{T}_{11} & \cdots & w_{21n_h}\dot{T}_{n_h1} \\ \vdots & & \vdots \\ w_{211}\dot{T}_{1n_s} & \cdots & w_{21n_h}\dot{T}_{n_hn_s} \\ w_{221}\dot{T}_{11} & \cdots & w_{22n_h}\dot{T}_{n_h1} \\ \vdots & & \vdots \\ w_{221}\dot{T}_{1n_s} & \cdots & w_{22n_h}\dot{T}_{n_hn_s} \\ w_{231}\dot{T}_{11} & \cdots & w_{23n_h}\dot{T}_{n_h1} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ w_{2n_{out}1}\dot{T}_{1n_s} & \cdots & w_{2n_{out}n_h}\dot{T}_{n_hn_s} \end{bmatrix} \tag{3.56}$$

$$YT = \begin{bmatrix} T_{11} & \cdots & T_{n_h1} & 0 & \cdots & 0 & & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \cdots\cdots & \vdots & & \vdots \\ T_{1n_s} & \cdots & T_{n_hn_s} & 0 & \cdots & 0 & & 0 & \cdots & 0 \\ 0 & \cdots & 0 & T_{11} & \cdots & T_{n_h1} & & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \cdots\cdots & \vdots & & \vdots \\ 0 & \cdots & 0 & T_{1n_s} & \cdots & T_{n_hn_s} & & 0 & \cdots & 0 \\ & \vdots & & & \vdots & & & & \vdots & \\ 0 & \cdots & 0 & 0 & \cdots & 0 & & T_{11} & \cdots & T_{n_h1} \\ \vdots & & \vdots & \vdots & & \vdots & \cdots\cdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & & T_{1n_s} & \cdots & T_{n_hn_s} \end{bmatrix} \tag{3.57}$$

$$YI = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \vdots & & & \vdots \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 1 & 0 & \cdots & 0 \\ & \vdots & & \vdots & \\ & \vdots & & \vdots & \\ 0 & \cdots & \cdots & 0 & 1 \\ \vdots & & & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \tag{3.58}$$

The T's should be calculated as the first thing since they are utilized in the first three sub matrices 3.55, 3.56 and 3.57. The activation function in 3.39 must be known in order to calculate the T's and has been chosen to be.

$$activationfunction(x) = T(x) = tanh(x) \tag{3.59}$$

since it has a simple derivative.

$$\dot{T} = 1 - T^2 \tag{3.60}$$

A matrix the size of the complete derivative matrix 3.54 can be created and initialized with zeros , where the matrix is mostly zero (YT and YI, see equation 3.57 and 3.58), with the following calls.

```
Result.Reallocate(nodo,nh*(ni+1+no)+no);
xd1          = nh*(ni+1)+1;
xd2          = xd1+no*nh;
Zero(Result,1,xd1,nodo,no*(nh+1));
```

Where

`ni`	=	n_{in}
`no`	=	n_{out}
`nh`	=	n_h
`nodo`	=	$n_s n_{in}$

The T's and the $\dot{T}$'s can be calculated easily with the following C++ code using the matrix library A.5.

```
I              = Transpose(W1*X);
RowAdd(I,B1);
TI             = tanh(I);                    (3.61)
TDI            = 1.0 - (TI%TI);
```

The sub matrices 3.55, 3.56, 3.57 and 3.58 are then calculated and inserted in the zero matrix. The following C++ code illustrate how this can be achieved using the matrix library A.5.

```
WL      = ExpandMatrix(W2,ns,ni);
XL      = GrowMatrix(Transpose(X),nh,no);
TL      = GrowMatrix(ExpandMatrix(TDI,1,ni),1,no);
WR      = ExpandMatrix(W2,ns,1);
TR      = GrowMatrix(TDI,1,no);
OneVec  = Ones(ns,1);

InsertMatrix((WL % TL % XL) | (WR % TR),    // YWTX | YWT
            dYnndWwv, 1, 1);

for(i=0;i<no;i++)
{
    // YT elements
    InsertMatrix(TI,dYnndWwv,i*ns+1,xd1+i*nh);
    // YI elements
    InsertMatrix(OneVec,dYnndWwv,i*ns+1,xd2+i);
}
```

3.5.8 Programming the Derivative with Respect to the Inputs

Programming the derivative with respect to the inputs 3.53 can also be done more easily by writing up the complete matrix as with 3.54. Such a matrix can be obtained by inserting 3.47 into 3.53.

In this case it will be an advantage to split the matrix up into nh matrices so that each matrix contains elements corresponding to a term in the sum in 3.47. In other

words, k is held constant in each matrix and increased by one for the next matrix, etc.

3.53 can then be written like this.

$$\frac{\partial Y_{nn}}{\partial X} = \sum_{k=1}^{nh} \begin{bmatrix} w_{21k}\dot{T}_{k1}w_{111} & \cdots & w_{21k}\dot{T}_{k1}w_{11n_{in}} \\ \vdots & & \vdots \\ w_{21k}\dot{T}_{kn_s}w_{111} & \cdots & w_{21k}\dot{T}_{kn_s}w_{11n_{in}} \\ w_{22k}\dot{T}_{k1}w_{111} & \cdots & w_{22k}\dot{T}_{k1}w_{11n_{in}} \\ \vdots & & \vdots \\ w_{22k}\dot{T}_{kn_s}w_{111} & \cdots & w_{22k}\dot{T}_{kn_s}w_{11n_{in}} \\ \vdots & & \vdots \\ w_{2n_{out}k}\dot{T}_{k1}w_{111} & \cdots & w_{2n_{out}k}\dot{T}_{k1}w_{11n_{in}} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ w_{2n_{out}k}\dot{T}_{kn_s}w_{111} & \cdots & w_{2n_{out}k}\dot{T}_{kn_s}w_{11n_{in}} \end{bmatrix} \tag{3.62}$$

The $T's$ can be calculated as before as in 3.61 and the following C++ code generates a large matrix WTW consisting of all the terms in 3.62 concatenated after each other. The special `ColSums()` function from the matrix library (See appendix A.5 table A.9) is then utilized to sum up the appropriate elements.

```
WL      = ExpandMatrix(W2,ns,ni);
I       = Transpose(W1*X);
RowAdd(I,HBias);
TI      = tanh(I);
TDI     = 1.0 - (TI%TI);
TM      = GrowMatrix(ExpandMatrix(TDI,1,ni),no,1);
WR      = GrowMatrix(Mat2RowVec(HWeights),ns*no,1);
WTW     = WL%TM%WR;

dYnndX  = ColSums(WTW,ni);
```

3.6 Conclusions

3.6.1 Predictive Training Algorithm

A predictive training algorithm was developed which takes the predicted errors up until a specified horizon into the cost function when training the neural networks. This proved to be very valuable when training dynamic neural network models since it reduces the chances of ending up with an unstable model and helps increase the accuracy of the neural network prediction. A demonstration of the predictive training algorithm is also given in which the predictive training algorithm clearly outperforms the standard one step ahead training algorithm commonly utilized.

It is especially valuable for training neural networks utilized in a predictive controller since it usually has a short prediction horizon. The prediction horizon given to the predictive training algorithm is usually short too because of the large memory consumption and training time necessary when training with long prediction horizons. A very small neural network prediction error within the prediction horizon is however almost always the result of training neural network with the predictive training algorithm and this is important for the predictive controller.

Chapter 4

Mean Value Engine Modelling (MVEM)

4.1 MVEM Introduction

The Mean Value Engine Model is one of the most widely used types of dynamic engine models. It models, as the name implies, the mean value of gross variables in an engine. This means that the fluctuations coming from the engine pumping are not taken into account, but the average value of the various signals like intake manifold pressure, port air mass flow and intake manifold temperature are modelled. The figure 4.1 illustrates what kind of signal the MVEM is modelling.

The blue signal is the measured throttle air mass flow and it fluctuates quite intensively as a result of the engine pumping. The points of the red curve have been generated as the mean value of 5 samples forwards and backwards of the original signal at the points of the signal of the blue curve at the same time. A simpler but still informative signal is obtained in this way. This makes it possible to model the engine utilizing relatively simple thermodynamics and flow models.

The derivation of the MVEM model will not be presented here in this dissertation, but the reader is referred to [24] and [12] for a complete derivation of the MVEM model.

This chapter begins with a presentation of two commonly used MVEM models. The isothermal MVEM (IMVEM) and the adiabatic MVEM (AMVEM). The IMVEM assumes that the temperature in the intake manifold stays constant and is therefore of one order less than the AMVEM which also models the temperature variation.

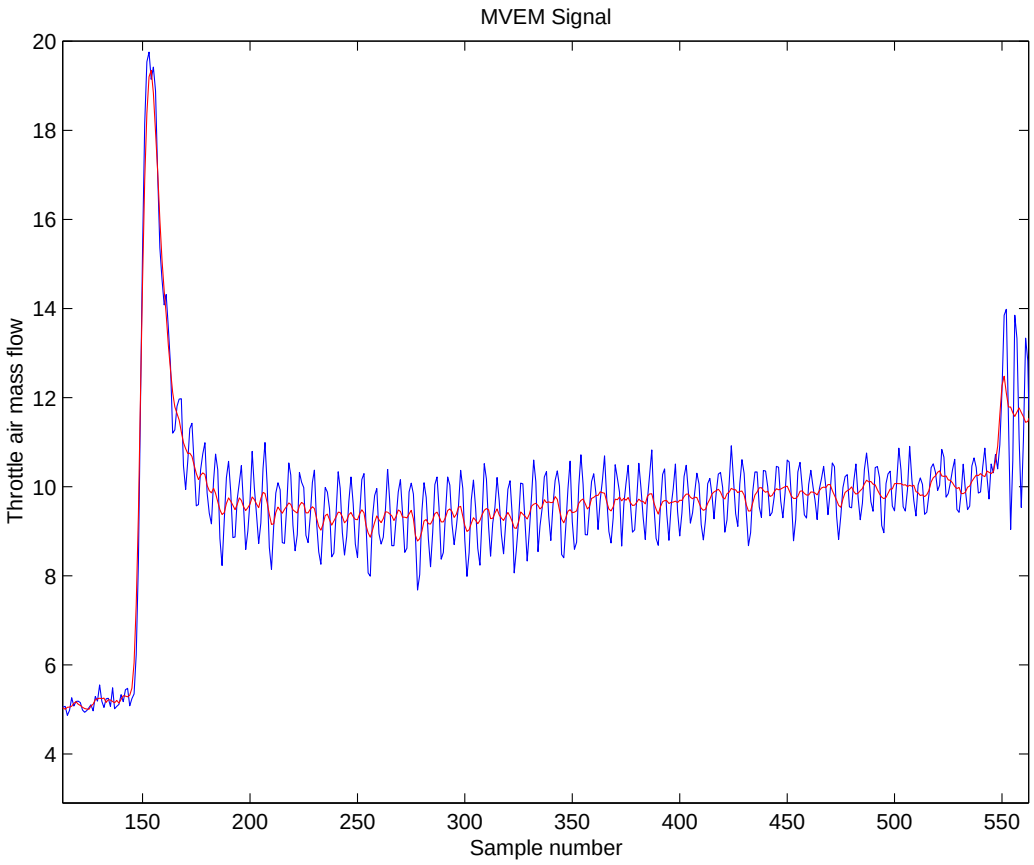


Figure 4.1: The throttle air mass flow and its MVEM mean value signal

The chapter then continues with an analysis of the problems related to the process of training a neural network MVEM model and presents the result achieved.

4.2 The ECG Test Engine

This chapter will present curves and data coming from the Engine Control Groups (ECG) test engine which is a British Leyland 1.275 Liter SI engine without EGR fitted with a sequential electronic fuel injection system. The engine is fitted with a number of sensors hooked up to a PC with an AD and a DA card. The PC is capable of both gathering data and to send out control signals. The engine will be referred to as the ECG test engine in the following.

British Leyland Engine Specifications		
Element	Value	Unit
Displacement Volume	1.275	L
Intake Manifold Volume	3.1	L
Cylinders	4	

Table 4.1: British Leyland 1.275 L Engine Specifications

The engine specifications are listed in table 4.1.

4.3 Common MVEM equations

The two MVEM models (the isothermal (IMVEM) and the adiabatic(AMVEM)) have a several equations in common which will be presented in the following sections.

Some suggestions to an extension to the throttle air mass flow equation commonly used in MVEM engine modelling have been developed during the work with the training of neural network MVEM models. This was discovered because the neural networks need adequately exciting data to learn the full range of the engine. The experiments performed to obtain such data was tested with the standard MVEM equations to compare the outputs with those from the neural networks. The input data to the engine such as the throttle plate angle were moved around in way that forced the intake manifold pressure up around atmospheric pressure which is a problem for the standard MVEM throttle air mass flow equations since a gradient approaches infinity as the intake manifold pressure approaches atmospheric pressure. Some extensions to the throttle air mass flow equations were developed and will be presented in the following sections.

4.3.1 The Crank Shaft Speed

The most fundamental part of the MVEM models is the equation governing the crank shaft speed. It is simply based on power balance and expresses the crank shaft acceleration as a function of the power input from the fuel and the power loss to the load and the internal friction. This is seen in the following equation taken from the paper [10]:

$$\dot{n} = -\frac{1}{I_n}(P_f + P_p + P_b) + \frac{H_u}{I_n}\eta_i \dot{m}_f(t - \Delta\tau_d) \tag{4.1}$$

where

I_n	The moment of inertia of the rotating parts in the engine.	$[kgm^2]$.
P_f	Power lost to friction.	$[kW]$
P_p	Power lost to pumping losses.	$[kW]$
P_b	Power lost to the load.	$[kW]$
H_u	Fuel burn value.	$[\frac{kJ}{kg}]$
η_i	Thermal efficiency.	[]
$\dot{m}_f$	Fuel mass flow.	$[\frac{kg}{s}]$
$\Delta\tau_d$	Combustion time delay.	$[s]$

The loss function P_b is the load input to the engine and can be implemented to match a desired operating scenario.

The loss functions P_f and P_p are usually regressions based on data from engine measurements. The loss functions for the ECG test engine utilized for the experiments in this dissertation can be modelled by the following regression functions taken from the paper [10]:

$$P_f = 0.0135n^3 + 0.2720n^2 + 1.6730n \tag{4.2}$$

$$P_p = nP_i(0.2060n - 0.9690) \tag{4.3}$$

Where

P_i	The pressure in the intake manifold.	$[bar]$
n	The crank shaft speed.	$[krpm]$

The thermal efficiency η_i is also a regression and is currently modelled for the ECG test engine by the following polynomial.

$$\eta_i = (0.558(1 - 0.392n^{-0.360}))(0.827 + 0.528P_i - 0.392P_i^2) \tag{4.4}$$

The combustion delay can be described by the following equation.

$$\Delta\tau_d = \frac{60}{n}\left(1 + \frac{1}{n_{cyl}}\right) \tag{4.5}$$

Where

n_{cyl}	The number of cylinders in the engine.

4.3.2 The Port Air Mass Flow

The port air mass flow is described by the following equation (Speed Density) taken from the paper [10].

$$\dot{m}_{ap} = \sqrt{\frac{T_i}{T_a}} \frac{V_d}{120RT_i}(e_v P_i)n \tag{4.6}$$

Where

$\dot{m}_{ap}$	The port air mass flow.	$[\frac{kg}{s}]$
T_i	The intake manifold temperature.	$[K]$
T_a	The ambient temperature.	$[K]$
V_d	The displacement volume.	$[m^3]$
R	The gas constant (Here 287.4e-05)	$[\frac{J\ bar}{kg\ K\ Pa}]$
e_v	The volumetric efficiency.	[]

The volumetric efficiency e_v can be described by the following simple equation taken from the paper [10].

$$e_v P_i = s_i(n)P_i + y_i(n) \tag{4.7}$$

Where $s_i(n)$ and $y_i(n)$ are function of n the crank angle speed. The $s_i(n)$ and $y_i(n)$ functions are mostly constant for high speeds, but changes values as n decreases. The graphs in figure 4.2 and 4.3 show what the $s_i(n)$ and $y_i(n)$ functions look like for the ECG test engine.

The $s_i(n)$ and $y_i(n)$ functions have been modelled by polynomials for the ECG test engine based on stationary mapping data from the engine obtained during the experimental work for this dissertation. The port air mass flow is equal to the throttle air mass flow which is measured when the engine is running stationary and the volumetric efficiency can thus be found from equation 4.6. The polynomials looks like this.

$$s_i(n) = 0.037559n^4 - 0.449575n^3 + 1.959664n^2 - 3.666088n^1 + 3.331978 \tag{4.8}$$

$$y_i(n) = -0.026999n^4 + 0.325134n^3 - 1.437841n^2 + 2.797844n^1 - 2.203290 \tag{4.9}$$

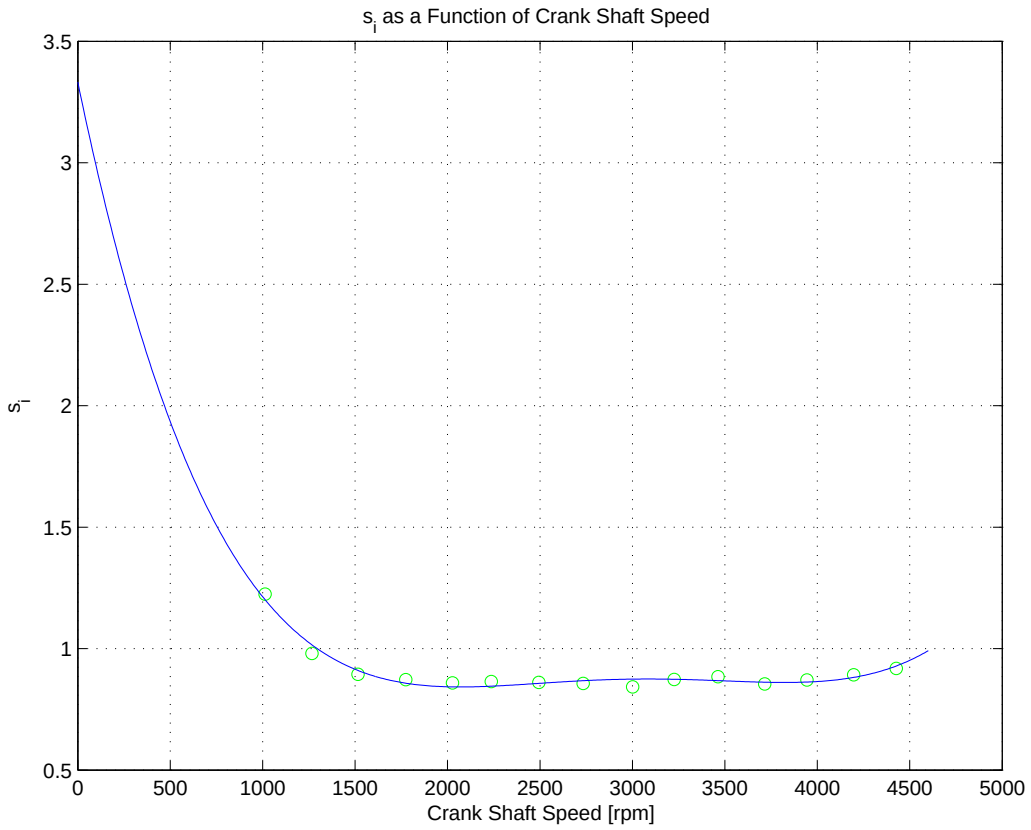


Figure 4.2: Si as a Function of n for the ECG Test Engine

4.3.3 The Throttle Air Mass Flow

The throttle air mass flow is described by the following equation taken from the paper [10].

$$\dot{m}_{at} = m_{at1}\frac{P_a}{\sqrt{T_a}}\beta_1(\alpha)\beta_2(P_r) + m_{at0} \tag{4.10}$$

Where

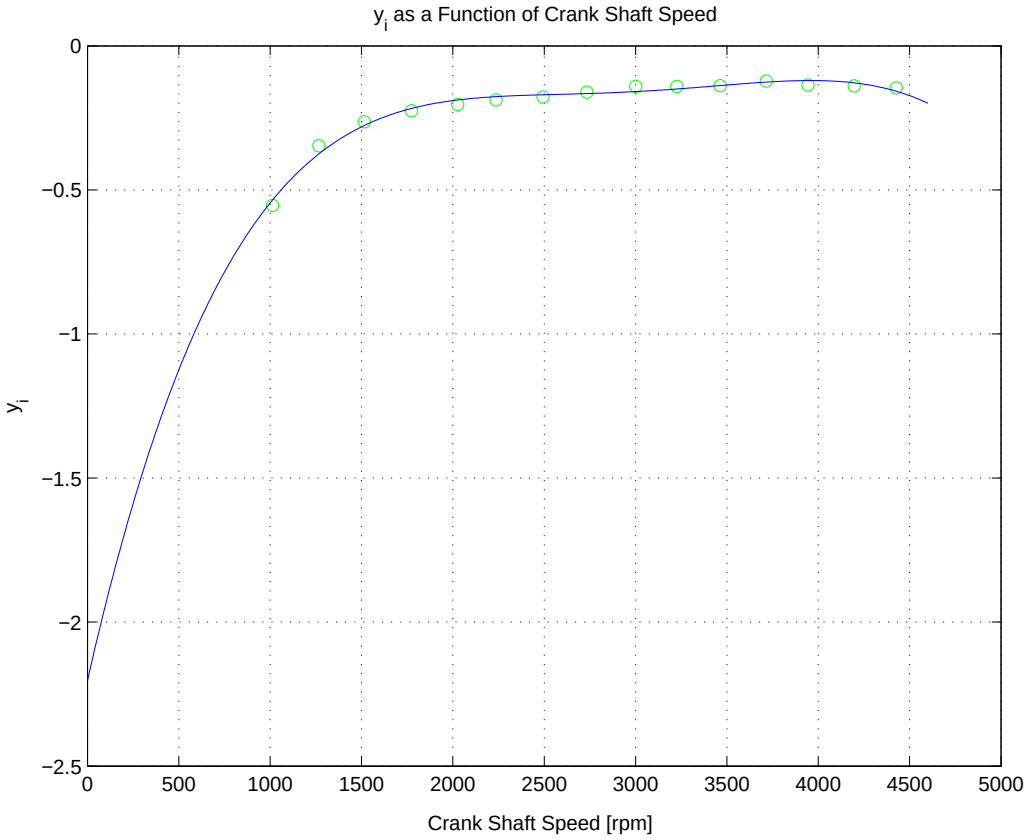


Figure 4.3: Yi as a Function of n for the ECG Test Engine

$\dot{m}_{at}$	The throttle air mass flow	$[\frac{kg}{s}]$.
m_{at1}, m_{at0}	Regression constants.	$[\frac{kg\sqrt{K}}{s\ bar}]$
P_a	Ambient pressure.	$[bar]$
α	Throttle plate angle.	$[^\circ]$
P_r	Pressure ratio $P_r = \frac{P_i}{P_a}$.	[]
β_1	Throttle flow dependency on the throttle plate angle.	[]
β_2	Throttle flow dependency on the pressure ratio.	[]

The throttle plate angle dependency can be described by the following function which is an approximation to the normalized open area.

$$\beta_1(\alpha) = 1 - cos(\alpha) - \left(\frac{\alpha_0}{2}\right)^2 \tag{4.11}$$

Where

α_0 The fully closed throttle plate angle (radians).

The pressure ratio dependency can be described by the following function.

$$\beta_2(P_r) = \begin{cases} 1 & P_r < P_c \\ \sqrt{1 - \left(\frac{P_r - P_c}{1 - P_c}\right)^2} & P_c \leq P_r \end{cases} \tag{4.12}$$

Where

P_c=0.4125 The critical pressure (Turbulent flow).

Equation 4.12 has a problem for $P_r \geq 1.0$ (which does occur at high output operating points) because the β_2 function has an infinitely large derivative at this point and because the function does not define what happens when the pressure in the intake manifold becomes larger than the ambient pressure.

This can happen for a very short time when the engine is accelerating and the throttle plate is opening rapidly. The fast opening of the throttle plate makes the air rush in from the outside. The burned gasses entering the intake manifold at the time when the intake valve and the exhaust valve are open at the same time increases the intake manifold pressure further. This can lead to a pressure in the intake manifold above the ambient pressure and the current β_2 function in equation 4.12 does not take that into account.

This is usually happens when the driver is "gorilla stomping" (depressing the accelerator pedal rapidly) as drivers normally do in cars with automatic transmission to make the car accelerate quickly.

Figure 4.4 shows a section from a data set where the pressure in the intake manifold exceeded the ambient pressure.

Continuous β_2 Extension

A possible extension to the β_2 function capable of dealing with higher than atmospheric pressure would be to mirror the function in the x-axis for values of $P_r \geq 1$ and possibly also scale it because the flow speed would most likely be of a different magnitude when it comes from the intake manifold out towards the ambient.

It could look something like this.

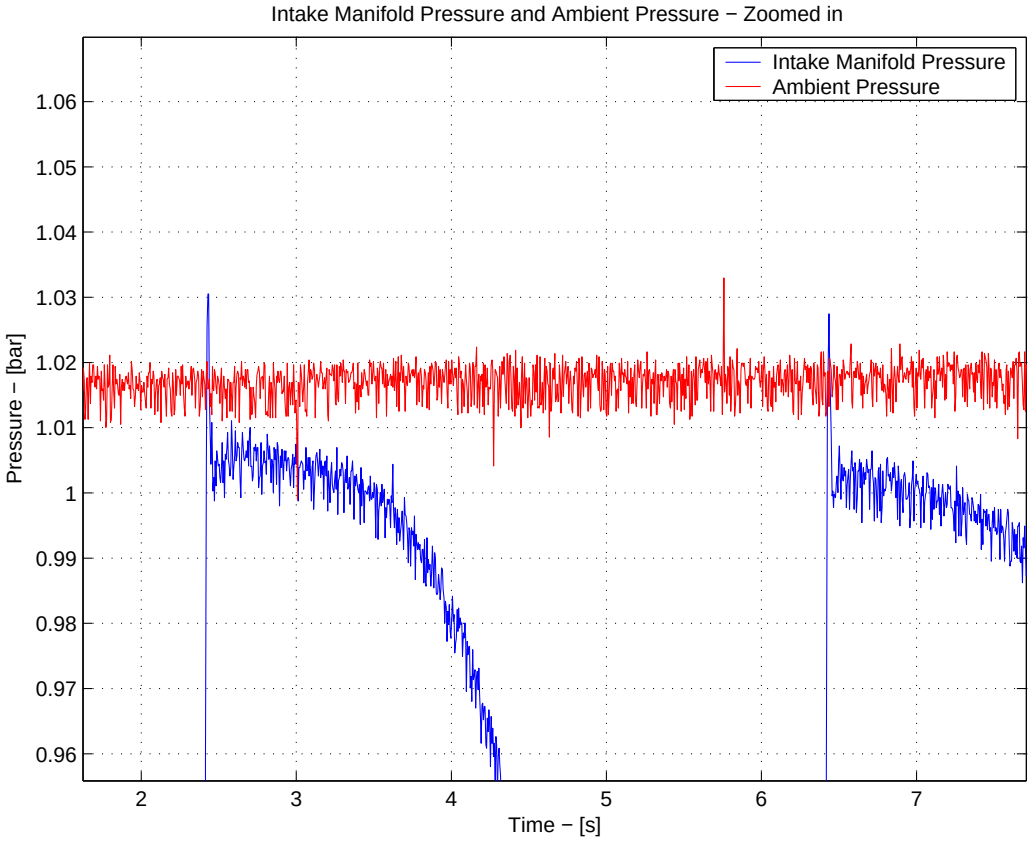


Figure 4.4: A section from a data set where the intake manifold pressure exceeds the ambient pressure

$$\beta MS_2(P_r) = \begin{cases} 1 & P_r < P_c \\ \sqrt{1 - \left(\frac{P_r - P_c}{1 - P_c}\right)^2} & P_c \leq P_r < 1.0 \\ K_\infty \sqrt{1 - \left(\frac{2 - P_r - P_c}{1 - P_c}\right)^2} & 1.0 \leq P_r \leq 1.0 + P_c \\ K_\infty & P_r > 1.0 + P_c \end{cases} \tag{4.13}$$

Figure 4.5 is a plot of the function in equation 4.13 with $K_\infty = -0.5$.

This function will however still have the problem with the infinitely large gradient at $P_r = 1.0$. But it models the throttle air mass flow for intake manifold pressures larger than ambient pressure.

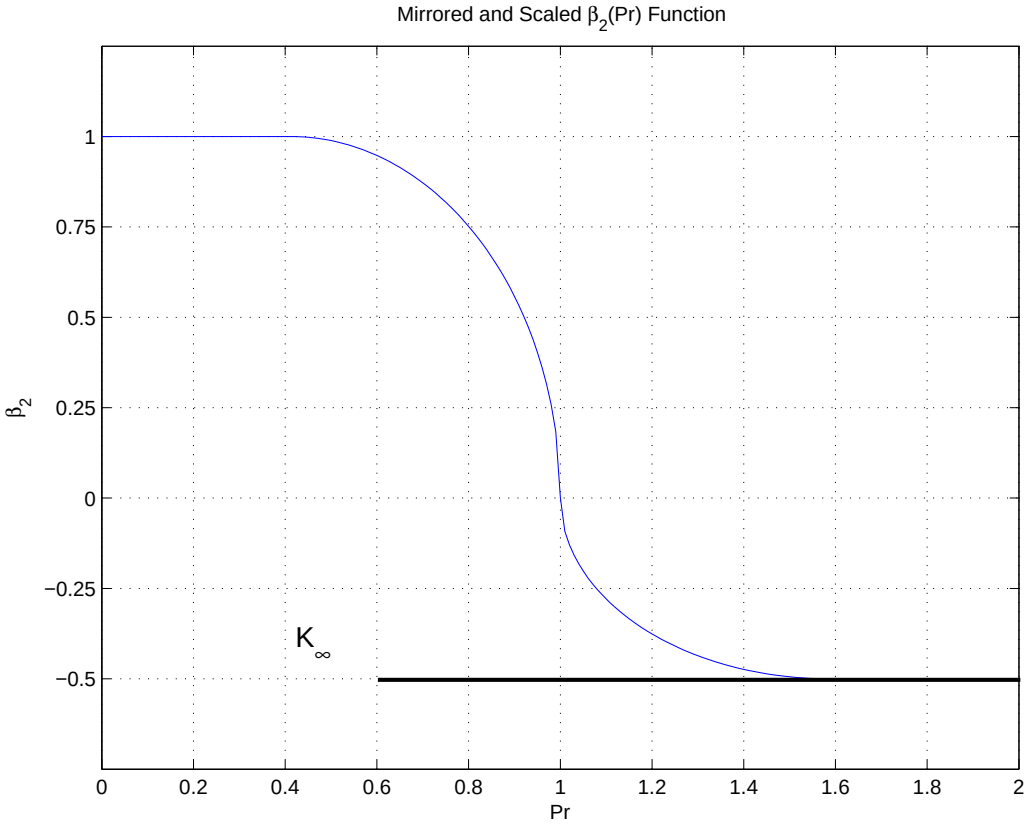


Figure 4.5: Mirrored and Scaled β_2 Function

Continuous β_2 Soft Extension

Another possible extension to the β_2 that eliminates the problem with the infinite gradient while still leaving the possibility to specify a different flow level at pressure ratios larger than 1 would be to extend the β_2 function continuously with a suitable function such as $tanh(x)$. More specifically including the regression constants necessary to fit the function continuously to the β_2 function.

$$Ext(Pr) = a\ tanh(b\ Pr + c) + d \tag{4.14}$$

The new softly extended β_2 function will be called $\beta_2 Soft$ and will have the following structure.

$$\beta_2 Soft(P_r) = \begin{cases} 1 & P_r < P_c \\ \sqrt{1 - \left(\frac{P_r - P_c}{1 - P_c}\right)^2} & P_c \leq P_r < Pr_0 \\ a\ tanh(b\ Pr + c) + d & Pr \geq Pr_0 \end{cases} \tag{4.15}$$

The following conditions are suggested for finding a,b,c and d.

1. Choose the crossing point where the tanh function takes over (Pr0). $\beta_2 Soft(Pr_0) = \beta_2(Pr_0)$
2. The β_2 should be continuous. $\frac{d\beta_2 Soft(Pr)}{dPr}(Pr_0 = \frac{d\beta_2(Pr}{dPr}(Pr_0)$
3. The value of $\beta_2 Soft(Pr)$ should go towards K_∞ as Pr goes towards ∞. $\beta_2 Soft(Pr) \rightarrow K_{infty}$ for $Pr \rightarrow \infty$.
4. The extension 4.14 should be symmetric around the Pr axis if possible. Otherwise choose d such that a solution exists.

Condition 3 means that

$$a = K_\infty \tag{4.16}$$

since $tanh(x) \rightarrow 1$ for $x \rightarrow \infty$.

Condition 1 and 2 yields the following equations for b and c.

$$b = \frac{P_c - Pr_0}{a(1 - P_c)\sqrt{(1 - P_c)^2 - (Pr_0 - P_c)^2}} \left(\left(atanh\left(\frac{1}{a}\sqrt{1 - \left(\frac{Pr_0 - P_c}{1 - P_c}\right)^2} - d \right) \right)^2 + 1 \right) \tag{4.17}$$

$$c = atanh\left(\frac{1}{a}\sqrt{1 - \left(\frac{Pr_0 - P_c}{1 - P_c}\right)^2} - d \right) - bPr_0 \tag{4.18}$$

If symmetry is desired (Condition 4) then

$$d = 0 \tag{4.19}$$

But symmetry is only possible if

$$\left| \frac{1}{a} \left(\sqrt{1 - \left(\frac{Pr_0 - P_c}{1 - P_c} \right)^2} - d \right) \right| < 1 \tag{4.20}$$

since this is the argument to the $atanh()$ function in the equation for b and c in 4.17 and 4.18.

This is especially a problem for small K_∞'s.

If symmetry is not possible then d should be chosen such that the argument to the $atanh()$ in equation 4.20 attains a chosen value numerically smaller than 1. This yields the following equations for a and d.

$$a = K_\infty - d \tag{4.21}$$

$$d = \frac{\sqrt{1 - \left(\frac{Pr_0 - p_c}{1 - P_c} \right)^2} - K_\infty atanh_0}{1 + atanh_0} \tag{4.22}$$

Where $atanh_0$ is the chosen value for the argument to the $atanh()$ function.

A graph of the soft extension to the $\beta_2()$ function can be seen in figure 4.6.

4.3.4 β_2 Extensions Comments

These suggestions is not currently used anywhere as far as the author knows, but is merely included because data from the test engine indicated that this could be a problem with the current $\beta_2()$ function when the data is put through the AMVEM equations in a Simulink simulation for verification. The simulation can go very wrong if the wrong integrator is chosen or take much longer than necessary.

The $\beta_2 Soft$ function in equation 4.15 is used throughout this work with $K_\infty = 0$ and $Pr_0 = 0.99$ to deal with intake manifold pressures larger than ambient pressure. K_∞ is not made negative in this work as it is not clear at this time whether or not this is the right way to model it and exactly what K_∞ should be. But the $\beta_2 Soft$ function improves simulation speed.

4.4 The Intake Manifold Equations

This is where the adiabatic and the IMVEM is different. The IMVEM uses only one state equation for the intake manifold pressure P_i and assumes that the intake

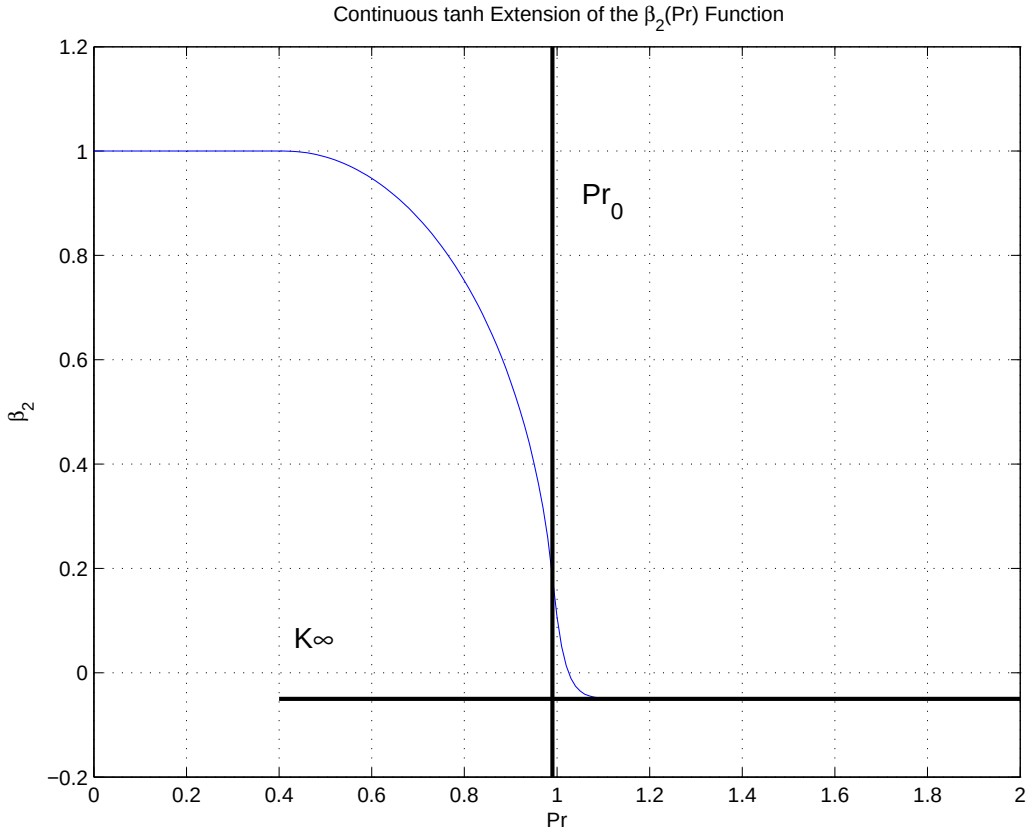


Figure 4.6: A Continuous tanh() Extension of the β_2 Function

manifold pressure stays constant.

4.4.1 Isothermal Equation

The isothermal intake pressure manifold equation taken from the paper [24] with the EGR term added has the following appearance.

$$\dot{P}_i = \frac{RT_i}{V_i}(\dot{m}_{at} + \dot{m}_{egr} - \dot{m}_{ap}) \tag{4.23}$$

Where

V_i	The intake manifold volume.	$[m^3]$
$\dot{m}_{egr}$	The Exhaust Gas Recycling (EGR) mass flow.	$[\frac{kg}{2}]$

4.4.2 Adiabatic Equations

There will be two state equations if the intake manifold temperature is not assumed to be constant. One for the pressure and one for the temperature. It assumed that no heat transfer takes place in the intake manifold. The equations are taken from the paper [10], but the heat capacity constants are assumed to be different for fresh air and exhaust gasses. The equations are.

$$\dot{P}_i = \frac{R_{mix} T_i}{V_i} \left(\frac{T_a c_{p,fresh}}{T_i c_{v,mix}} \dot{m}_{at} + \frac{T_{egr} c_{p,egr}}{T_i c_{v,mix}} \dot{m}_{egr} - \frac{c_{p,mix}}{c_{v,mix}} \dot{m}_{ap} \right) \tag{4.24}$$

$$\dot{T}_i = \frac{R_{mix} T_i^2}{P_i V_i} \left(\left(\frac{T_a c_{p,fresh}}{T_i c_{v,mix}} - 1 \right) \dot{m}_{at} + \left(\frac{T_{egr} c_{p,egr}}{T_i c_{v,mix}} - 1 \right) \dot{m}_{egr} + \left(1 - \frac{c_{p,mix}}{c_{v,mix}} \right) \dot{m}_{ap} \right) \tag{4.25}$$

Where

R_{mix}	Is the gas constant for the mixture of fresh air and exhaust gas.
$c_{p,fresh}$	Is the constant pressure specific heat constant for fresh air.
$c_{v,mix}$	Is the constant volume specific heat constant for the mixture gas.
T_{egr}	Is the temperature of the exhaust gas.
$c_{p,egr}$	Is the constant pressure specific heat constant for the exhaust gas.
$c_{p,mix}$	Is the constant pressure specific heat constant for the mixture gas.

The intake temperature state equation can also be replace by the mass conservation state equation and a temperature output equation as follows.

$$\dot{m}_i = \dot{m}_{at} + \dot{m}_{egr} - \dot{m}_{ap} \tag{4.26}$$

$$T_i = \frac{P_i V_i}{m_i R_{mix}} \tag{4.27}$$

Where

m_i	Is the mass of the air in the intake manifold $[kg]$

4.5 The AMVEM Accuracy in the Literature

The following sections describes the experiments reported in two papers ([24] and [44]) utilizing the the AMVEM model. The results in the papers give an idea of the accuracy of the AMVEM model.

4.5.1 Intake Manifold Modelling Accuracy

The IMVEM is insufficient since the temperature in the intake manifold does vary significantly during fast throttle movement and especially when there is EGR in the system. The exhaust gasses in the intake manifold contains so much energy that the temperature of the intake manifold no longer can be considered constant.

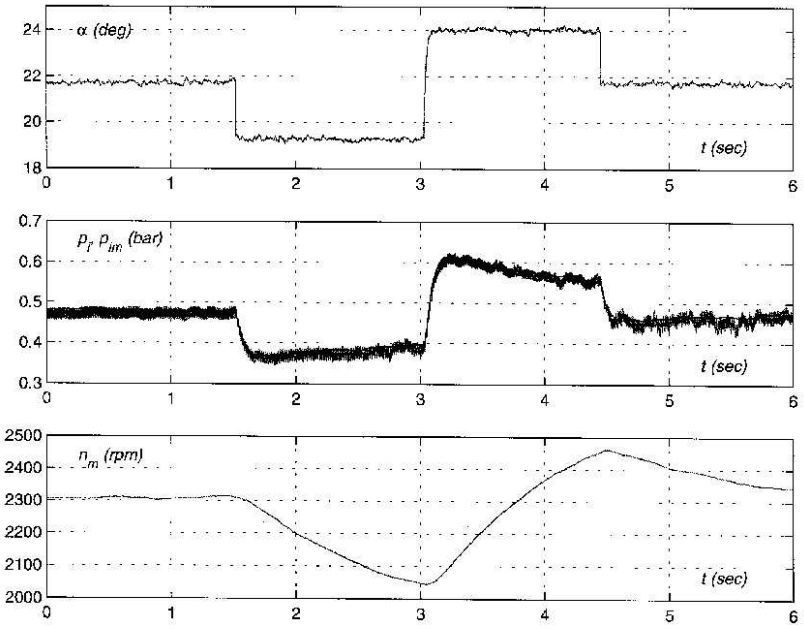
The AMVEM was also meant to compensate for the temperature changes caused by fast throttle movements and EGR by including the intake manifold temperature as a state.

The paper [24] analyzes the performance of the AMVEM on an SI engine with EGR by comparing the output of the AMVEM with the real signals. The authors concludes that AMVEM's signals follow the measurements well except for a small error in the temperature which the authors of that paper think is caused by heat transfer not modelled by the AMVEM.

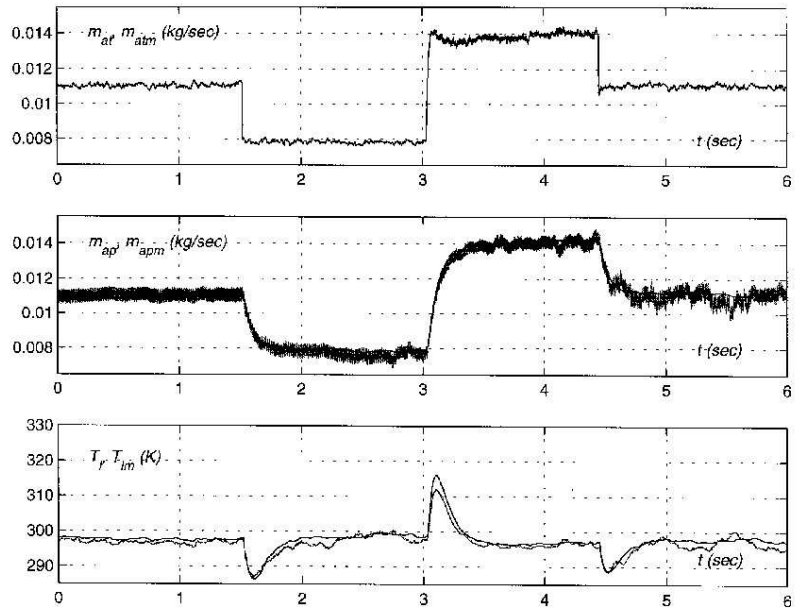
The graph in the paper [24] showing the simulation results and the measurements indicate that the pressure is modelled extremely well by the AMVEM. The temperature is also followed well by the model, but not at all as accurately as the pressure.

The test in this paper is only performed for one low load operating point of the engine, but is according to one of the authors of the paper [24] Elbert Hendricks is representative of the modelling capabilities over a large operating area.

The simulation results graphs from the paper [24] are copied here and shown in figure 4.7 and 4.8 for easy review.

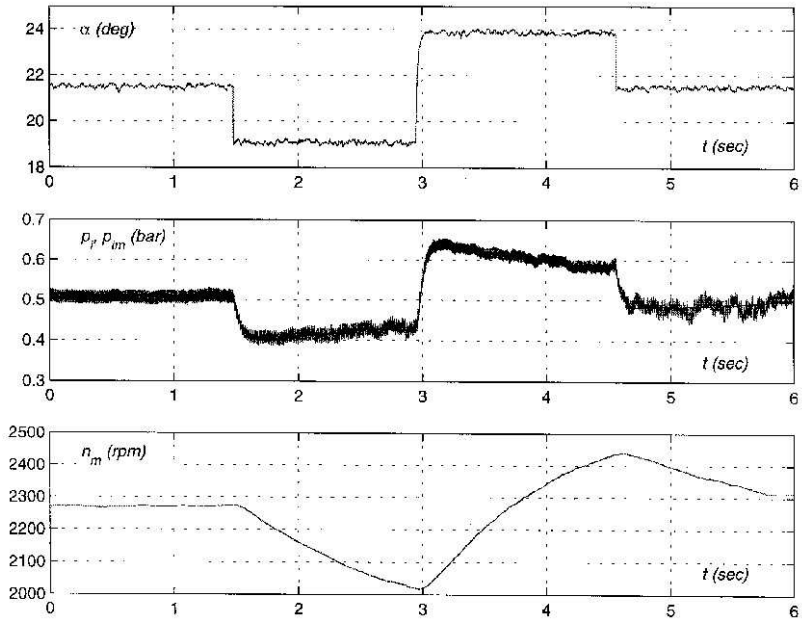


(a) AMVEM pressure

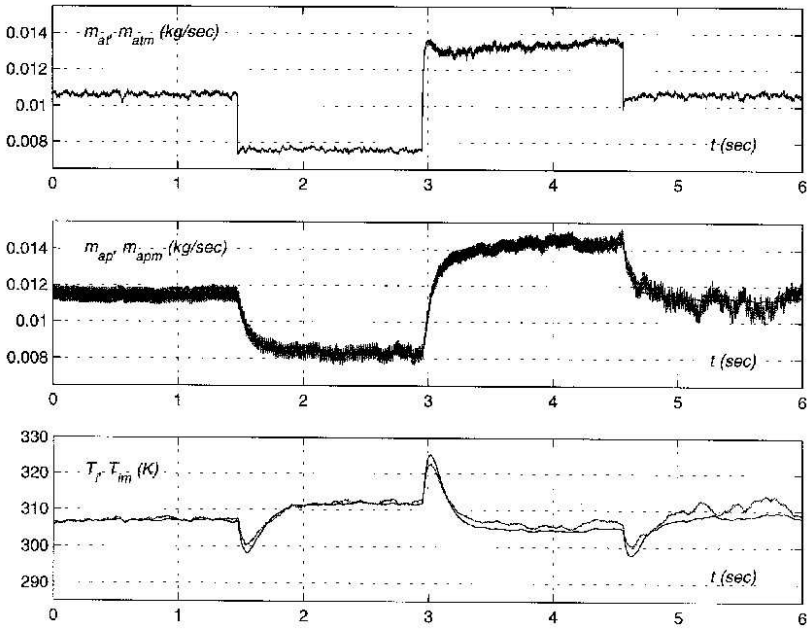


(b) AMVEM temperature

Figure 4.7: AMVEM Outputs Compared to a Measured SI Engines Signals - No EGR



(a) AMVEM pressure



(b) AMVEM temperature

Figure 4.8: AMVEM Outputs Compared to a Measured SI Engines Signals - EGR

4.5.2 Lambda Modelling Accuracy

The reason for the desire to model the intake manifold conditions very accurately, especially during transients, is the desire to fuel the engine stoichiometrically at all times to avoid pollution.

The catalyst used to clean the exhaust gasses require that the air fuel ratio (the lambda value) in the cylinder is very close to the stoichiometric ratio which is about 14.67 for gasoline. The lambda value is however normally normalized with respect to this stoichiometric value and should therefore always stay very close to 1.0 for optimal catalyst efficiency.

In this work it is desired to examine the performance of an advanced neural predictive controller and it is therefore necessary to have a model capable of predicting the lambda value accurately.

There are not many models of the lambda value available in the literature today to the knowledge of the author. There is however one interesting model using the AMVEM framework and the x, τ fuel film model (See [17]) can be found in the paper [44].

The paper does not specifically show the accuracy of the lambda value model, but applies the model in an H_{inf} controller setup for air fuel ratio control. The controller performs rather well. It keeps the air fuel ratio with $\pm 8\%$ of the stoichiometric value. This would indicate some level of accuracy.

The lambda value model is based on the AMVEM equations and is thus affected by the MVEM modelling errors. There should therefore be a possibility of improving this result by using a more accurate model which could be possible with a neural network model.

4.6 The AMVEM Accuracy in a Wider Operating Area

Neural network training requires adequately exciting training data in order to ensure that the neural network model will cover as large an operating area as possible.

A temperature modelling error by the AMVEM model when given inputs going

through a wider operating area than in the paper [24] mentioned previously was discovered during the work with neural network model training.

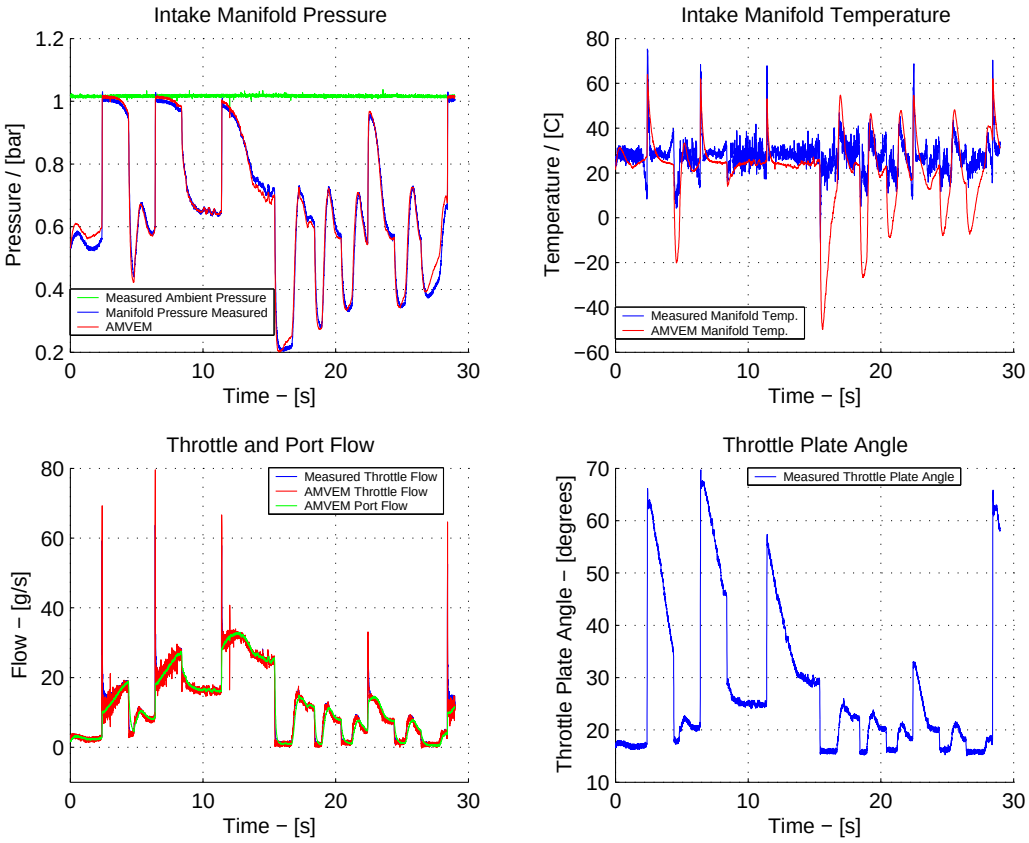


Figure 4.9: AMVEM Simulation and Measured Data

Figure 4.9 shows the large temperature errors made by the AMVEM model when the model is given aggressive throttle plate inputs as those also seen in the figure.

There are also some larger errors in the pressure and the throttle air mass flow. The throttle air mass flow errors are more clearly seen in figure 4.10 where a close up of a manifold filling spike can be seen. The throttle air mass flow error is only significant during the manifold filling spike, but this is however also a very important event since errors made here cause serious pollution levels during the frequent accelerations and decelerations made by a driver in larger cities.

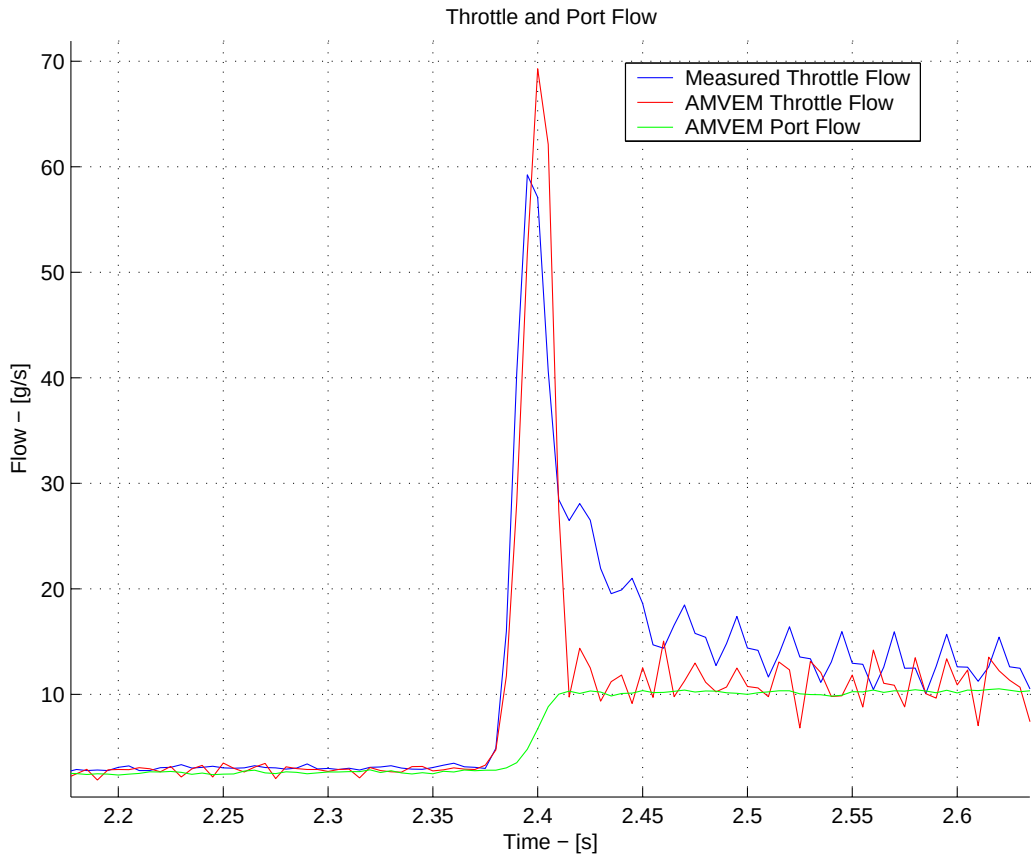


Figure 4.10: AMVEM Air Mass Flows and Measured Data

4.7 Neural MVEM modelling

4.7.1 Introduction

The AMVEM modelling framework makes many simplifications and assumptions to obtain a simple physical model capable of describing the most important physical aspects of an engine.

The AMVEM model does a very good job following the mean values of the engine signals, but not perfectly so. The authors of the paper [24] believe that the errors in the modelling of the intake manifold temperature is caused by heat transfer which is not modelled in the AMVEM.

One might argue that the AMVEM model is good enough, but as pollution con-

trol demands are ever increasing so is the problem of fulfilling them! And tighter control will be necessary in the future and thus also more accurate modelling and control.

The graphs in figure 4.9 and figure 4.10 show that an engine controlled by a system applying the AMVEM model would make large errors in the estimation of port air mass flow since the throttle air mass flow cannot be trusted during manifold filling spikes and because the intake temperature and pressure (especially the temperature) are wrong. This will cause large amounts of pollution from the engine when driving in a larger city since the engine will be in a transient state more often than not because of traffic lights and the intense traffic itself.

The estimation of the port air mass flow will however still have problems if the volumetric efficiency is not well described dynamically. The volumetric efficiency as previously described is modelled by polynomials regressions (equation 4.8 and 4.9) which are obtained from steady state engine mapping data. The port air mass flow and the throttle air mass flow are equal when the engine is running in steady state because of mass conservation. The port air mass flow is therefore not necessarily correct during transients. It is however normal to utilize steady state data for this since it is quite difficult to measure the port air mass flow.

The port air mass flow was however not available during this work and can thus not be verified or modelled by a neural network here.

It is however desired in this work to train a neural network model that can be utilized in the predictive controller described in the next chapter. This is attempted by training a neural network model of the entire engine, modelling it from the inputs (throttle plate angle command and fuel mass flow command) of the engine to the outputs (air fuel ratio and engine speed). This neural network model will then have to be able to internally describe the port air mass flow in order to keep the air fuel ratio in place. This was unfortunately not possible in this work as will be explained in the section about the neural network crankshaft speed and lambda model. The lambda model could not be made accurate enough as a single output dynamical neural network and it has thus been postponed for later work.

A neural network can if given an adequately exciting training data set extract all the information that has been disregarded in the AMVEM model (such as heat transfer and the large temperature errors) and also make up for the errors seen in the previous section in figure 4.9 and 4.10 and thus become a more accurate model.

This is what the following sections are about. Various subsystems of the AMVEM

are modelled by neural network models and the modelling performance is compared with the corresponding AMVEM subsystem if available (not for the lambda value for instance). This will give a picture of level of accuracy currently obtainable with the neural network models utilized in this work.

The AMVEM described in section 4.3 models EGR in the engine, but the ECG test engine that has been available to the author of this dissertation was not setup for EGR. The training of the neural network engine models will thus be compared to the AMVEM equations with the EGR part taken out. There are however still big errors, without EGR, in the AMVEM model as the graphs in the paper about the AMVEM modelling [24] and the wide operating area AMVEM simulation in figure 4.9 show.

4.7.2 Training and Test Data Set Generation

The training and test data for the neural network modelling was sampled at 200 Hz from the ECG Test Engine described in section 4.2.

The training of neural networks require adequately exciting input data in order to provide the neural network training algorithm with all the different input and output combinations necessary to extract all the system model information from the engine. This is similar to the necessity of having three equations when a unique solution for three variables are desired.

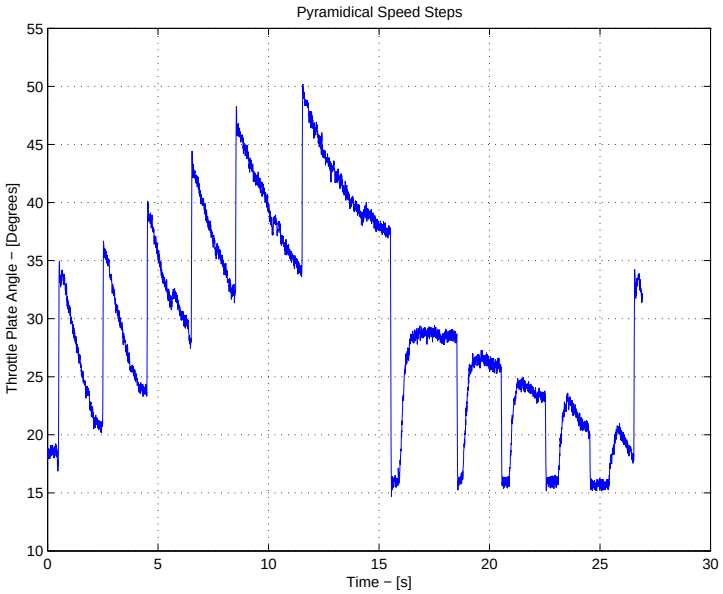
The dynomometer at the ECG Test Engine can be put in a constant load mode where it will allow the crank shaft speed to change freely. This mode is selected for the data gathering process in order to make the engine speed move from idle speed to about 4500 rpm which is about as high as the engine will go without too many problems. The load is however not constant since the dynomometer is unable to do keep it. This is actually what is desired in this case since it will provide data from many more operating points.

A cranks shaft speed controller was developed for the engine control system at the test engine to make sure the engine would not over speed and to be able to control to some extent the operating state of the engine. A sequencing program moves the engine from about idle speed to about 4000 rpm and down again by changing the reference speed to the controller according to a table. The data is logged simultaneously. The speed reference given to the speed controller is moved from low to high in both a regular pyramidical step pattern as shown in figure 4.13(a), but also in a more varied fashion as shown in figure 4.13(b).

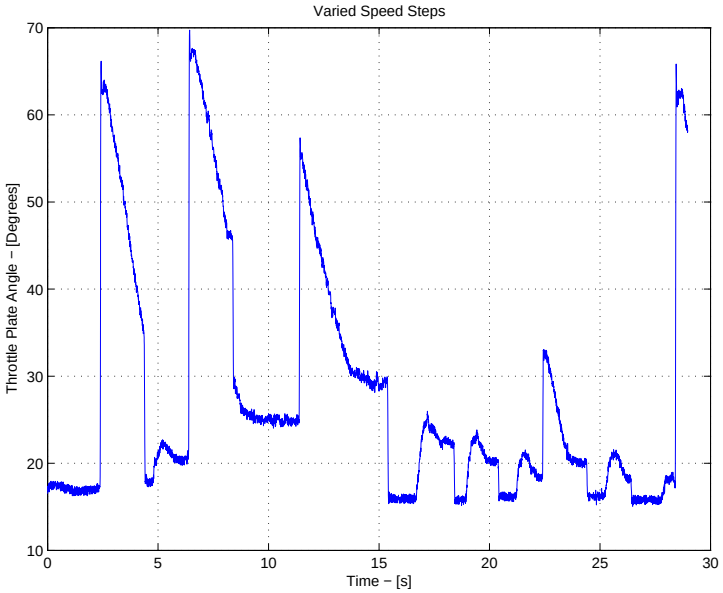
Data was also gathered by simply moving the throttle plate angle in steps from almost closed to the largest possible angle depending on the load from the dynomometer and down again to almost closed. The throttle plate was moved in both a regular pyramidical step pattern and in a more varied fashion.

Figure 4.11(a) shows how the throttle plate angle was typically moved, when utilizing the speed controller, in the regular pyramidical step pattern and figure 4.11(b) shows how it was typically moved in the more varied fashion.

Figure 4.12(a) shows how the throttle plate angle was typically moved, when simply controlling the throttle plate directly, in the regular pyramidical step pattern and figure 4.12(b) shows how it was typically moved in the more varied fashion.

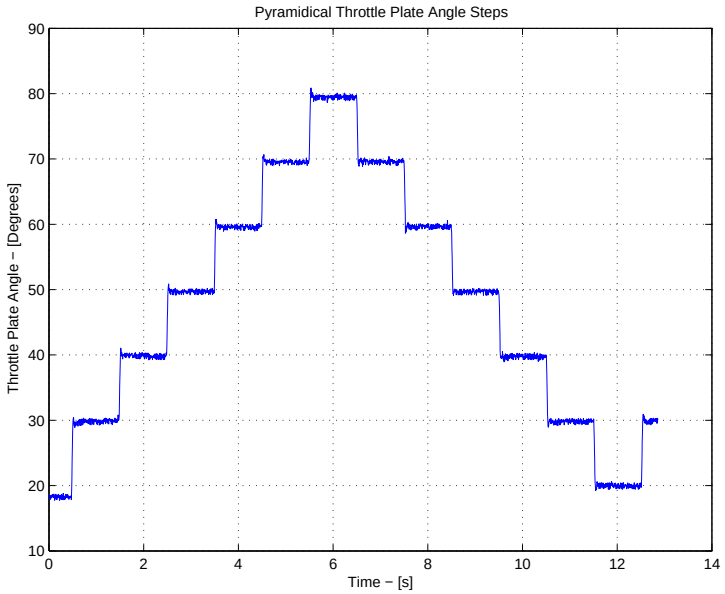


(a) Throttle Plate Angle - Pyramidical Speed Steps

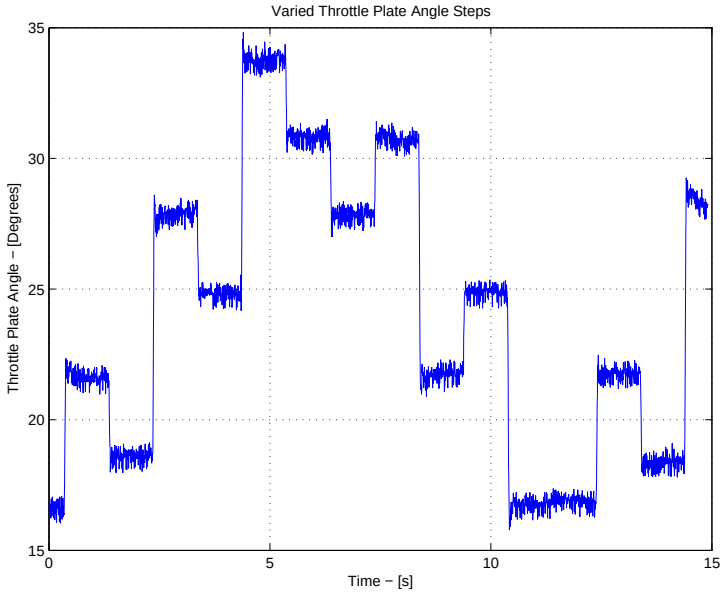


(b) Throttle Plate Angle - Varied Speed Steps

Figure 4.11: Throttle Plate Angle - Speed Steps

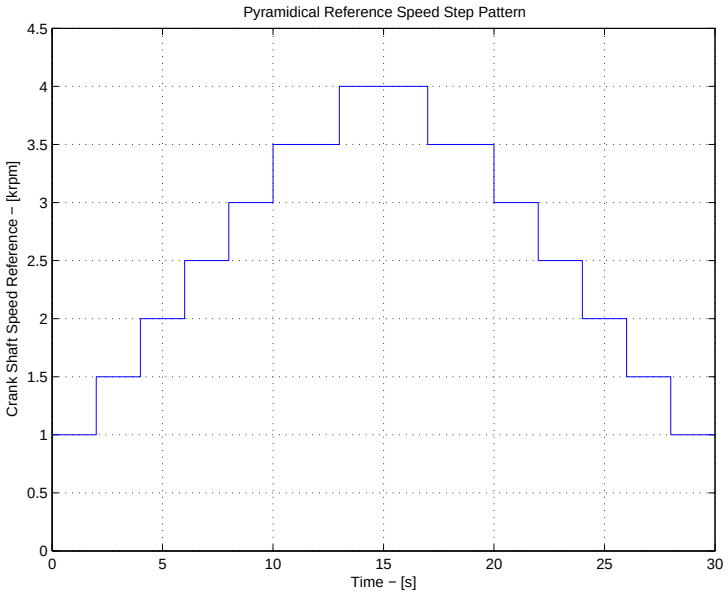


(a) Throttle Plate Angle - Pyramidical Throttle Plate Angle Steps

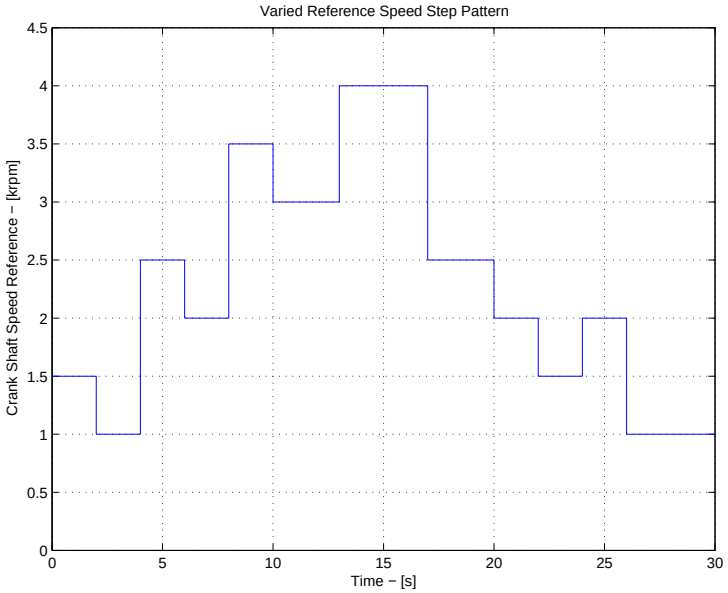


(b) Throttle Plate Angle - Varied Throttle Plate Angle Steps

Figure 4.12: Throttle Plate Angle - Throttle Plate Angle Steps



(a) Pyramidical Reference Speed Step Pattern



(b) Varied Reference Speed Step Pattern

Figure 4.13: Reference Speed Step Patterns

4.7.3 Fallout Removal

Some of the sensors utilized for data gathering would occasionally fail and provide a completely wrong output far from the real one. This is usually seen in the data sets as large spikes (see the blue spikes in figure 4.14).

This is very undesirable since it severely disturbs the neural network model training process. The value at a sensor fall out is usually far away from the real value and is completely uncorrelated with the system state measured. This will have a great effect on the cost function for the neural network training algorithm since the error is squared, although the magnitude of the effect also depends on the number of fallouts, their distance from the real value and the length of the data set.

A filter has been developed to automatically detect such fallouts and to replace the erroneous values with more correct values. A demonstration of the filter can be seen in figure 4.14.

The filter works by calculating the difference signal for the signal in mind as follows (MATLAB syntax). The signal with the fallouts will be called x.

```
xd = [0 x(2:end)-x(1:end-1)];
```
(4.28)

A noise limit value, nl, is utilized to find the indices where there might be a sensor fallout. It is done by finding all the indices where

```
exi = find(abs(xd)>nl)
```
(4.29)

.
The difference will typically be much larger at the indices where there is a sensor fallout.

The same is done to a smoothed "support" signal. The smoothing is performed by replacing each point in the "support" signal by the mean value of a specified number of points around that point. The calculation of a difference signal for the smoothed "support" signal is then performed. The location of the indices where the difference signal for the smoothed "support" signal is larger than a specified value as with exi in equation 4.29 are found. Those indices will be called zdi.

The idea is then to replace the points in the signal x with the the values from a heavily smoothed version of the signal x at the indices in exi that are not in zdi. Since if the indices in exi are also in zdi then the explanation for the large value of the difference signal for x at those indices is most likely that it is a naturally

occurring step caused by a step in some inputs signal.

The "support" signal chosen for the data here is the throttle plate angle since it is a driving input and signals like the throttle air mass flow, intake manifold pressure and intake manifold temperature respond with a step when a step is applied to the throttle plate angle.

The source code for the fall out removal MATLAB function utilized in this work can be found in appendix A.9 and on the source code appendix CD in the folder ToolBoxes/NeuralSysID.

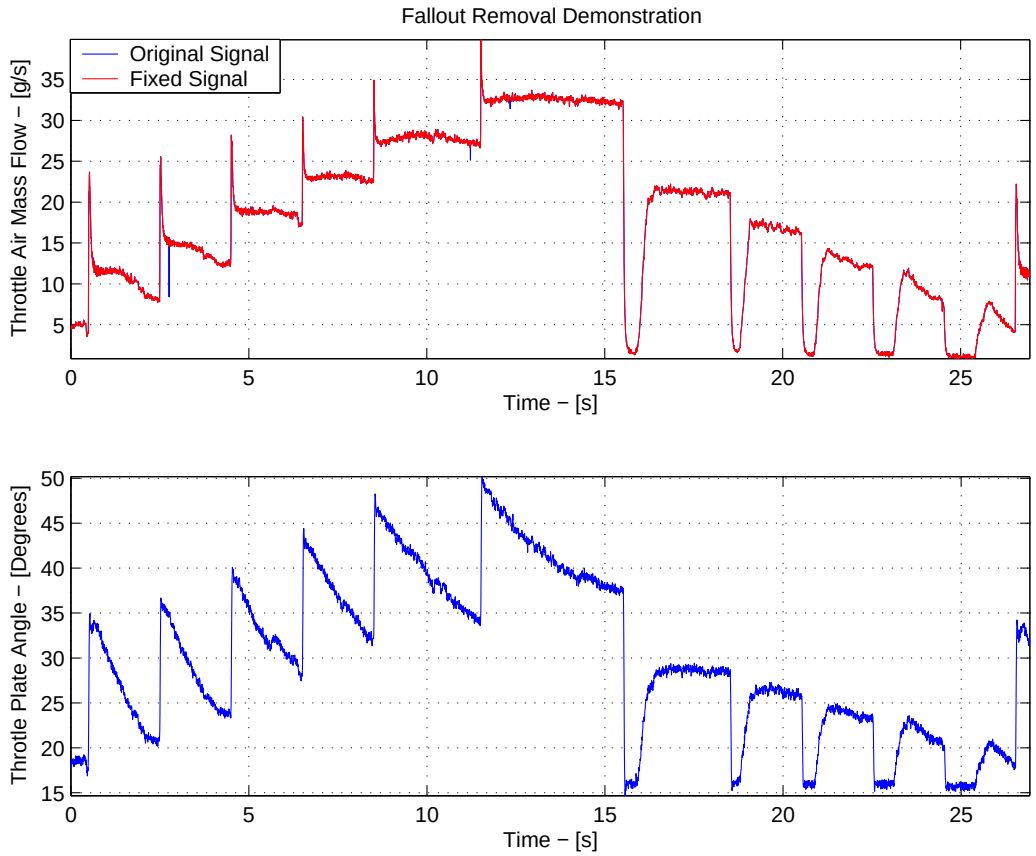


Figure 4.14: Fall Out Removal

4.7.4 Temperature Sensor Problems

All neural network models that include the temperature as an input or as an output are somewhat difficult to handle since an adequately fast temperature measurement in the intake manifold is quite difficult to obtain. A new experimental temperature sensor was constructed by Elbert Hendricks at DTU (Danish Technical University) for this experiment, but it can not be described here because it is subject to a patent application. The new temperature sensor is however not at this time completely reliable and it is plausible that this has made the training of the neural network subsystems utilizing the temperature signal difficult. The high bandwidth of the new temperature sensor, which is in the order of a 1 ms time constant, still has some problems with noise. The small time constant makes it subject to turbulence in the intake manifold air stream. The accuracy of the new fast temperature sensor has not yet been confirmed.

The neural network modelling results presented here in this chapter should therefore be viewed with this in mind.

4.8 Engine Subsystems Neural Network Modelling

The AMVEM consist of the first two subsystems in the following list, but the two next subsystem are also vital for accurate pollution control. The neural network subsystem modelling results for the subsystems listed below will be presented in the following.

1. The throttle air mass flow. $\dot{m}_{at}$, equation 4.10.

2. The intake manifold pressure and temperature. (P_i and T_i, equation 4.24 and 4.25).

3. The crank shaft speed. (n, equation 4.1).

4. The air fuel ratio (Lambda).

The following neural network models are all trained utilizing the predictive training algorithm described in section 3.3. The type of neural network utilized is the single hidden layer neural network described in section 1.3.

4.8.1 Input Output Signals Symbol List

The names listed in table 4.2 are the short names that will be utilized for the engine signals in the neural network model descriptions.

Input Output Signal Name List		
Name	**Signal**	**Unit**
α_c	Throttle Plate Angle Command.	$[^\circ]$
$\dot{m}_{fc}$	Fuel Air Mass Flow Command.	$[\frac{g}{s}]$
$\dot{m}_f$	Actual Fuel Mass Flow.	$[\frac{g}{s}]$
α	Throttle Plate Angle.	$[^\circ]$
$\dot{m}_{at}$	Throttle Air Mass Flow.	$[\frac{g}{s}]$
$\dot{m}_{ap}$	Port Air Mass Flow.	$[\frac{g}{s}]$
P_a	Ambient Pressure.	$[bar]$
T_a	Ambient Temperature.	$[C]$
P_i	Intake Manifold Pressure.	$[bar]$
T_i	Intake Manifold Temperature.	$[C]$
$P_r = \frac{P_i}{P_a}$	Intake Manifold Pressure Ratio.	[]
N	Crank Shaft Speed.	$[kRPM]$
$\lambda = \frac{\dot{m}_{ap}}{\dot{m}_f}$	Normalized Air Fuel Ratio.	[]

Table 4.2: Input Output Signal Name List

4.8.2 Throttle Air Mass Flow Modelling

The following dynamic neural network structure was the best one found.

$$\dot{m}_{at,k+1} = NN(\dot{m}_{at,k}, \dot{m}_{at,k-1}, \alpha_k, \alpha_{k-1}, P_{r,k}, P_{r,k-1}) \tag{4.30}$$

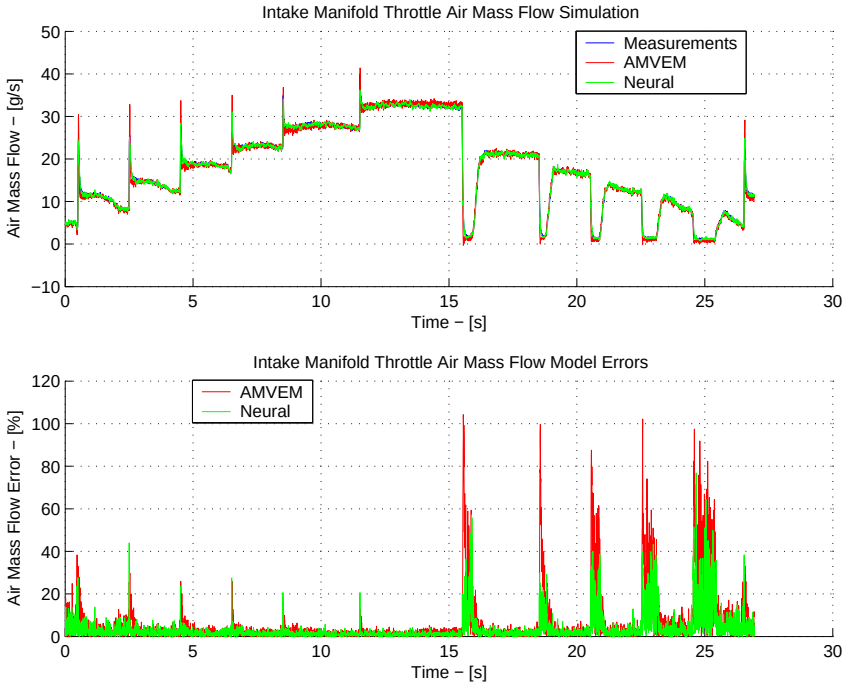
The inputs chosen for this neural network throttle air mass flow model are $\dot{m}_{at}, \alpha$ and P_r since α and P_r are physical factors determining the throttle air mass flow (see AMVEM throttle air mass flow equation 4.10). $\dot{m}_{at}$ is a feed back input to make the neural network model dynamical. The number of hidden neurons for the neural networks in the following throttle air mass flow simulations is 6.

A simulation of the throttle air mass flow neural network model utilizing input signals from the training data set can be seen in figure 4.15(a) and for a test data set in figure 4.15(b).

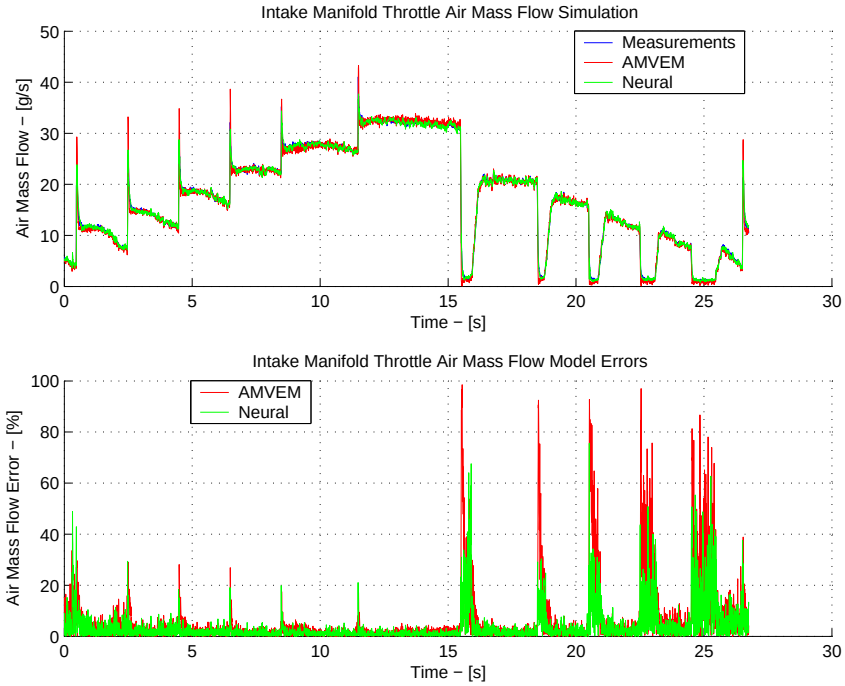
The ambient temperature and the intake manifold temperature was not utilized as inputs even though the AMVEM equations do so (See equation 4.10). This is because it would not make much sense for these experiments because of the small change in ambient temperature which will make it impossible to obtain adequately exciting training data for the neural network. Furthermore, the inclusion of the intake manifold temperature does not improve the accuracy of the neural network model, but only adds to the complexity (more weights) of it.

The accuracy of the neural network model is in many places better than the standard AMVEM throttle air mass flow model (See equation 4.10). A close up on some of the manifold filling and emptying spikes for test set simulation in figure 4.15(b) illustrates this in figure 4.17.

The improved accuracy however only holds for test sets where the throttle plate inputs are similar (pyramidical type of speed steps for the results in figure 4.15(a) and 4.15(b)). Figure 4.16(a) shows a complete simulation on a test data set generated with the varied type of throttle plate angle steps as shown in figure 4.12(b) and figure 4.16(b) shows a close up on some of the manifold filling and emptying spikes in figure 4.16(a). The accuracy of the neural network model is here worse than the AMVEM model. This is however the best neural network throttle air mass flow model that could be found.

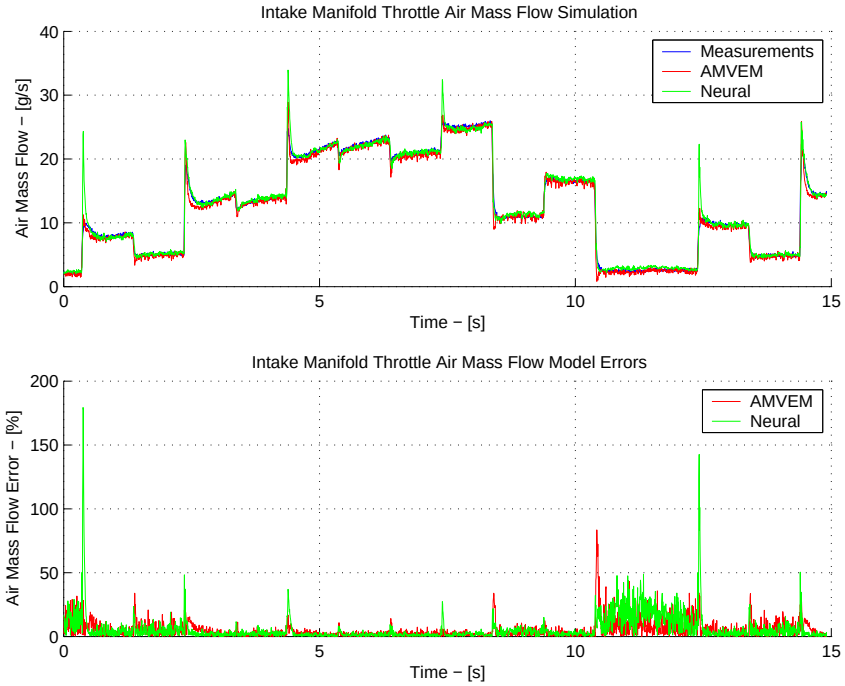


(a) Throttle Air Mass Flow - Training Set Simulation

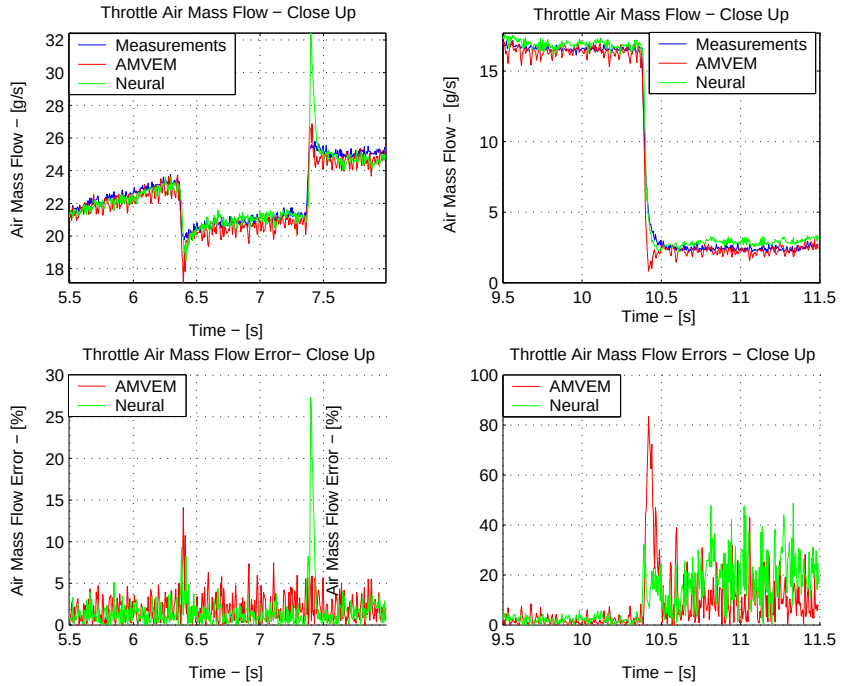


(b) Throttle Air Mass Flow - Test Set Simulation

Figure 4.15: Throttle Air Mass Flow Simulations



(a) Throttle Air Mass Flow - Test Set Simulation (Worse)



(b) Throttle Air Mass Flow - Test Set Simulation (Worse) - Close Up

Figure 4.16: Throttle Air Mass Flow Simulations

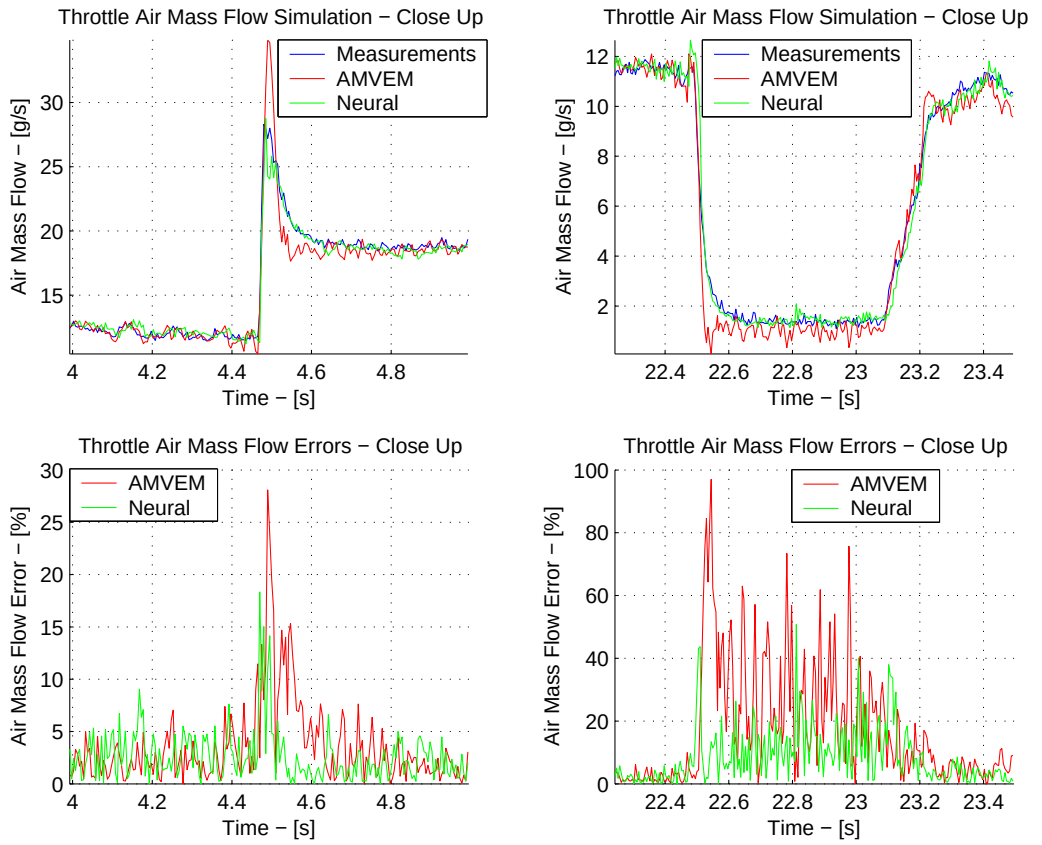


Figure 4.17: A Close Up on the Manifold Filling and Emptying Spikes for the Test Data Set Simulation

4.8.3 Lambda Modelling

The λ signal seems to get disturbed by some unknown source and has made id difficult to obtain a decent neural network model. However, the neural network model found in this work seems to, interestingly enough, be able to disregard those possible errors.

Figure 4.18 shows 30λ, α and $\dot{m}_{fc}$ plotted on top of each other. The graph shows that there is a peculiar large spike in the λ signal between 5.5 s and 5.8 s, but there are no large throttle plate movements or fuel mass flow commands in the that period. It is possible that this is a sensor error, but it could also be a naturally occurring phenomenon which needs more information to explain than the author has been able to find. One possible explanation could be failing fuel injectors

since that would temporarily increase the air fuel ratio as the spike shows in figure 4.18. The neural network modelling results achieved are however still presented here in this section since the neural network model in some cases seems to be able to disregard this phenomenon.

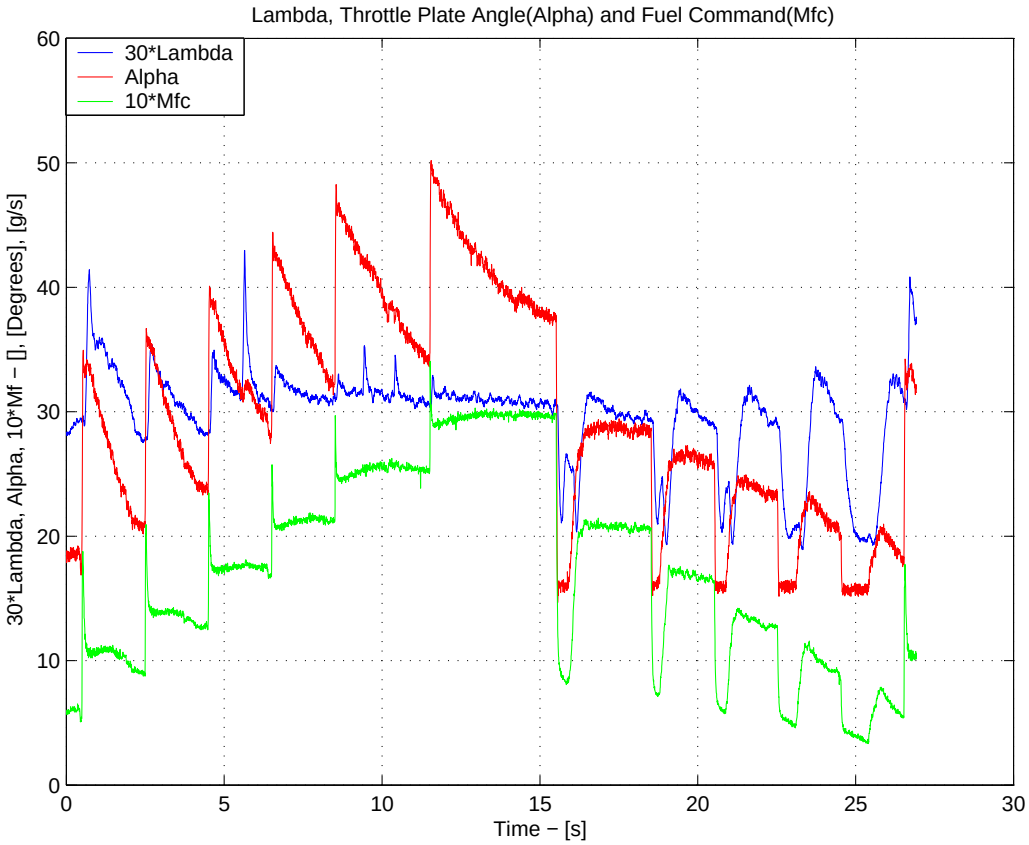


Figure 4.18: Lambda Sensor Error

The neural network λ model will not be compared to an AMVEM model since there are no well established AMVEM model of the air fuel ratio as far as the author is concerned. The neural network model output will thus only be compared to the measurements from the ECG Test Engine.

The best dynamic neural network configuration found in this work appears as follows.

$$\lambda_{k+1} = NN(\lambda_k, \lambda_{k-1}, \alpha_{c,k}, \alpha_{c,k-1}, \dot{m}_{fc,k}, P_{i,k}, T_{i,k}) \tag{4.31}$$

The inputs $\lambda, \alpha_c, \dot{m}_{fc}, P_i, T_i$ have been chosen since α_c, P_i and T_i are physically factors in determining the air mass flow (see the throttle flow equation 4.10 and the port flow equation 4.6). P_a and T_a are also factors, but are not included in this work since it was not possible to control those factors when gathering training data. Furthermore, the signal $\dot{m}_{fc}$ is clearly a factor in determining λ the air fuel ratio since it is the fuelling command. λ is a feed back input to make the neural network model dynamical. The number of hidden neurons for the neural networks in the following lambda simulations is 5.

Figure 4.19 shows simulations of the the neural network λ model for the training set and three test sets utilizing the inputs from each data set respectively.

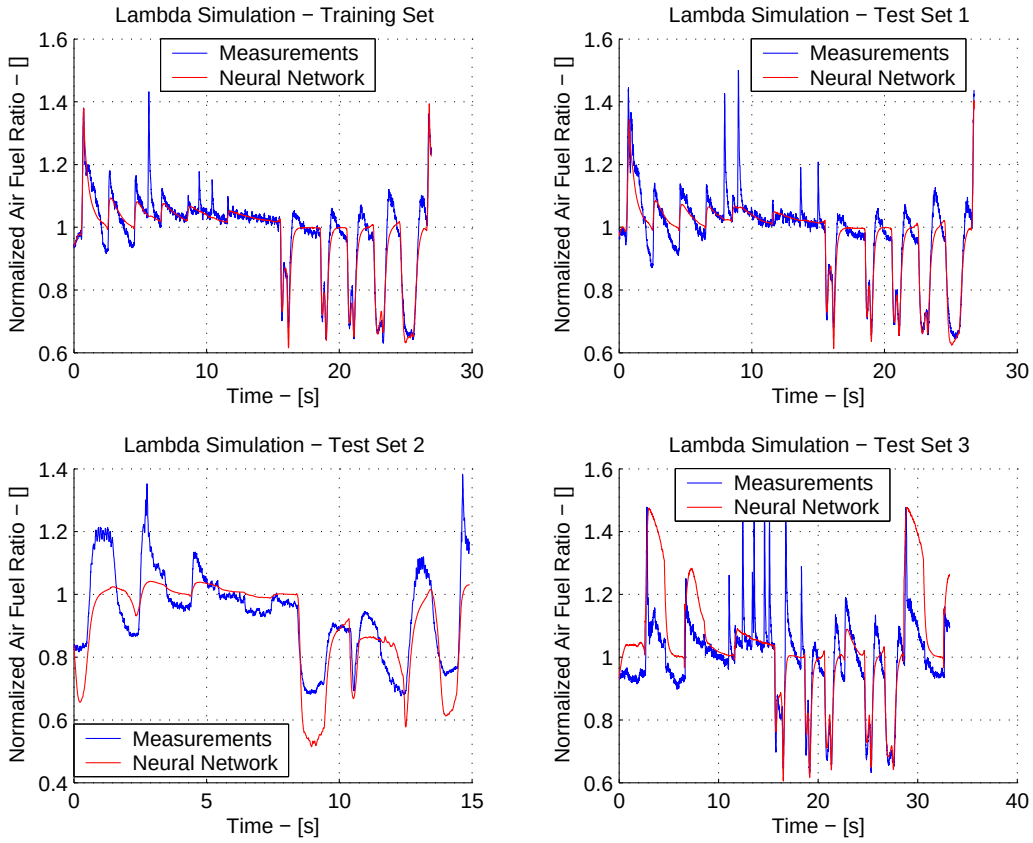


Figure 4.19: Lambda Simulations - Training and Test Sets

The neural network λ model is not performing very well according to the measurements. The errors are visibly far too large for the model to be of any use. The

utilization of the temperature signal in this neural network model and the possible λ sensor error could be the reason why this model is not currently working.

4.8.4 Cranks Shaft Speed and Lambda MIMO Modelling

The MIMO N and λ model intended for the MIMO neural predictive controller developed and described in the next chapter will not be presented here as it has not yet been possible to make such a neural network model accurate enough. The simulations of the achieved λ neural network model shows that there might be some sensors problems that has to be fixed before this is possible. The intake manifold temperature sensor is also causing some problems since its accuracy has not yet been confirmed and it is vital information for a lambda model. This MIMO model will be postponed for later research.

4.8.5 Intake Manifold Pressure Modelling

The best dynamic neural network P_i model found has the following structure.

$$P_{i,k+1} = NN(P_{i,k}, \alpha_k, N_k, T_{i,k}) \tag{4.32}$$

The inputs P_i, T_i and α are the physical factors determining the throttle air mass flow (see equation 4.10). P_a and T_a is not utilized here due to the lack of control with ambient parameters. N helps the neural network describe the port air mass flow (see equation 4.6). The neural network then has the necessary parameters to describe to incoming flow and the outgoing flow from the intake manifold. This, as can be seen from the intake manifold pressure equation 4.24, are the factors that are necessary to describe the pressure derivative. The number of hidden neurons for the neural networks in the following intake manifold pressure simulations is 5.

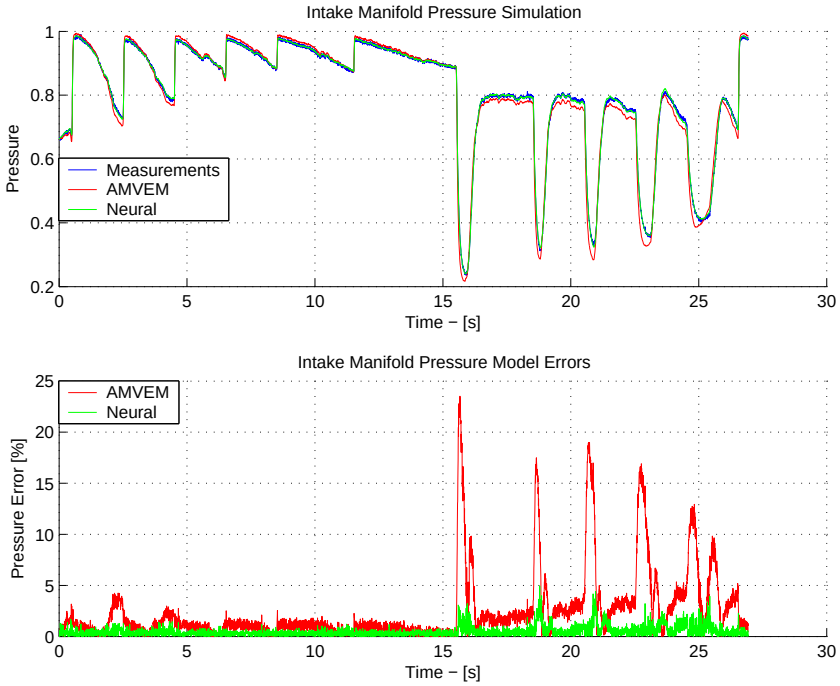
Figure 4.20(a) shows a neural network P_i model simulation utilizing the inputs from the training data set. The accuracy is quite good and clearly better than the AMVEM model.

Figure 4.20(b) shows a simulation utilizing inputs from a test data set generated with a similar throttle plate angle movement pattern as for the training set which is the pyramidical speed step pattern as seen in figure 4.11(a). The accuracy in this test data set is also very good.

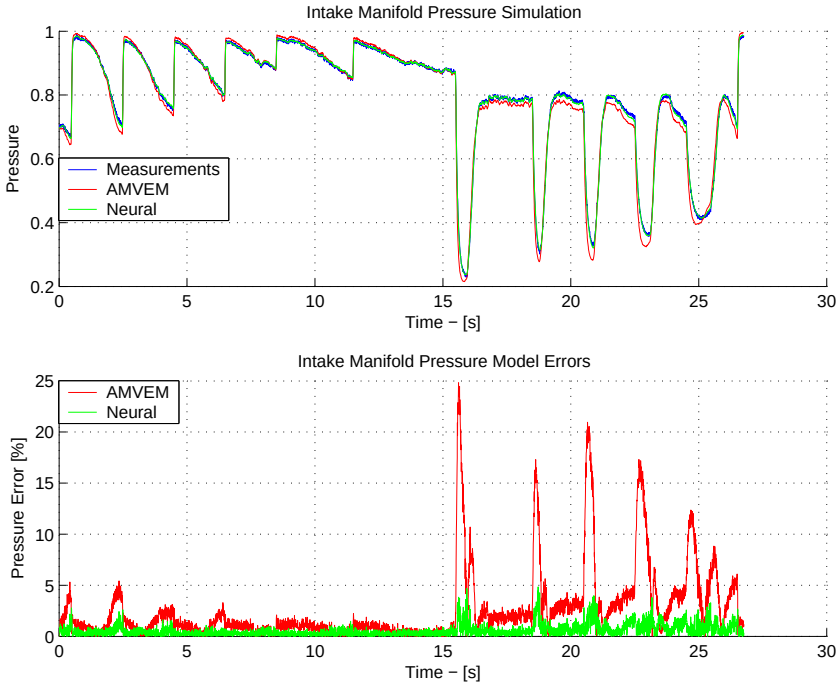
The neural network P_i model works in general better on all test data sets generated with the pyramidical type of speed step pattern (as in figure 4.11(a)) compared to

the AMVEM model.

Figure 4.21 shows a simulation of the neural network P_i model utilizing the inputs from a test data set generated with a varied type of throttle plate angle step pattern (as in figure 4.12(b)). The model does not perform as well in this case, but still well, on the test data sets generated with the varied type of throttle plate angle steps pattern. It is no longer clear for this kind of test data sets which model is best. But the overall performance on all data sets shows that the neural network P_i model is more accurate than the AMVEM model.



(a) Intake Manifold Pressure - Training Set Simulation



(b) Intake Manifold Pressure - Test Set Simulation

Figure 4.20: Intake Manifold Pressure Simulations - Better Than AMVEM

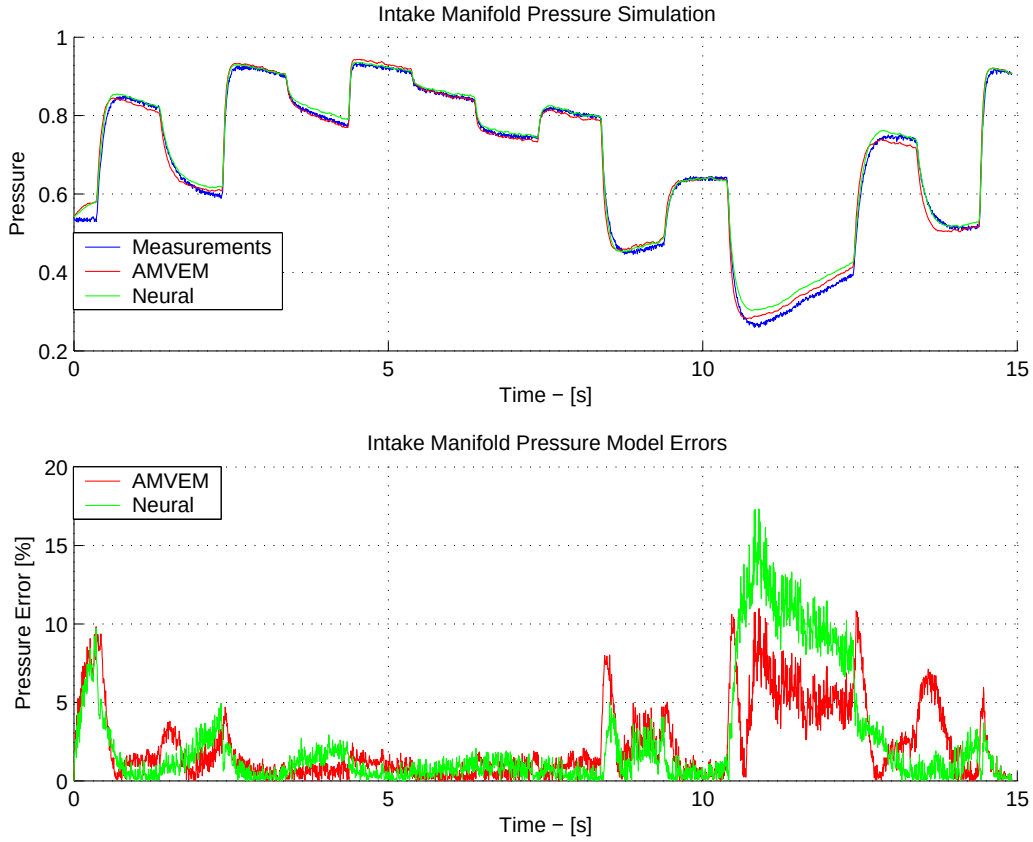


Figure 4.21: Intake Manifold Pressure Simulation - Worse

4.8.6 Intake Manifold Temperature Modelling

The best dynamic neural network temperature model structure found is similar to the one for the pressure model with the exception that the temperature and pressure are exchanged. It appears as follows.

$$T_{i,k+1} = NN(T_{i,k}, \alpha_k, N_k, P_{i,k}) \tag{4.33}$$

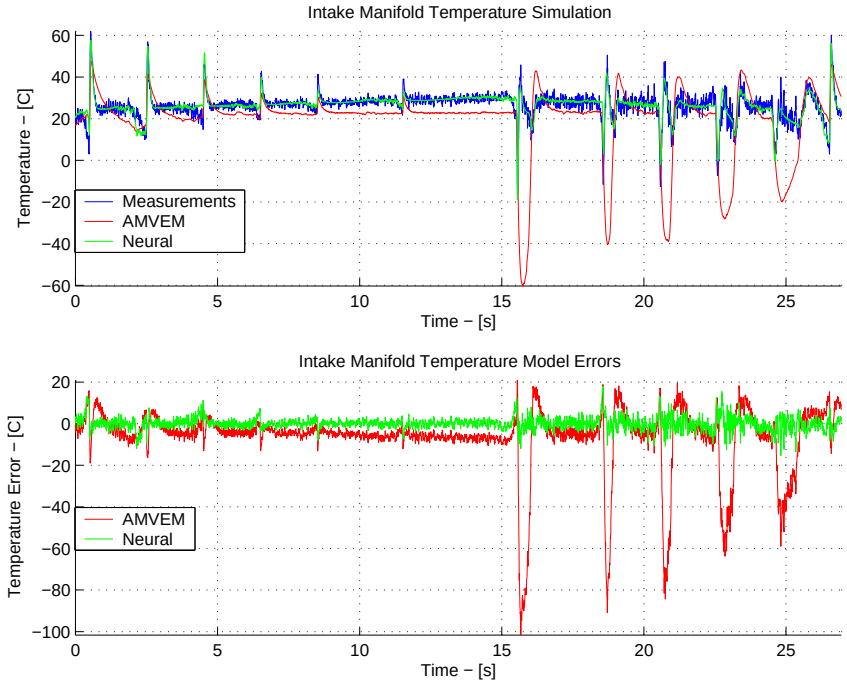
The number of hidden neurons for the neural networks in the following intake manifold temperature simulations is 5.

Figure 4.22(a) shows a neural network T_i model simulation utilizing the inputs from the training data set. The performance is clearly better than the AMVEM models temperature output.

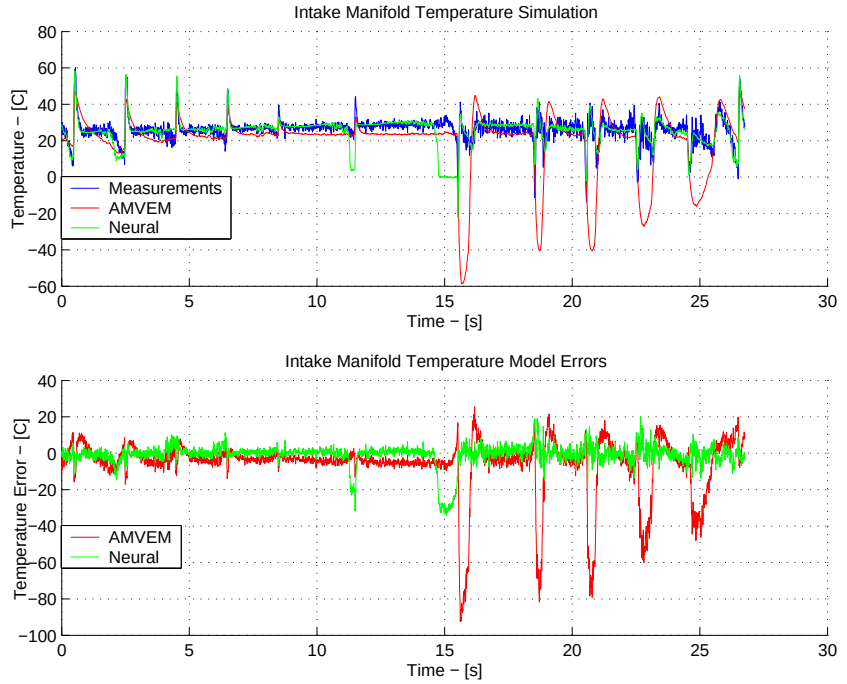
Figure 4.22(b) and 4.23 shows a simulation utilizing the inputs from some test data sets. The performance of the neural network model for first test data set simulation in figure 4.22(b) is for the most part ok and much better than the AMVEM models temperature output except for a large divergence around the time 15 s. This is however not the case for the test set simulation in figure 4.23. The performance of the neural network model is here about equal to that of the AMVEM model.

The training data set utilized for the simulation in figure 4.22(a) and the test data set utilized for the simulation in figure 4.22(b) are both generated with the pyramidical speed step type of throttle plate angle input pattern as shown in figure 4.11(a).

The reduced performance of the neural network model in figure 4.23 must be explained by the difference in the generation of the training data set (Figure 4.22(a)) and the test data set in figure 4.23. The test data set utilized for the test set simulation in figure 4.23 is generated with the varied type of throttle plate angle inputs as shown in figure 4.12(b) and that data set must contain some information not seen in the training data set.



(a) Intake Manifold Temperature - Training Set Simulation



(b) Intake Manifold Temperature - Test Set Simulation

Figure 4.22: Intake Manifold Temperature Simulations - Better Than AMVEM

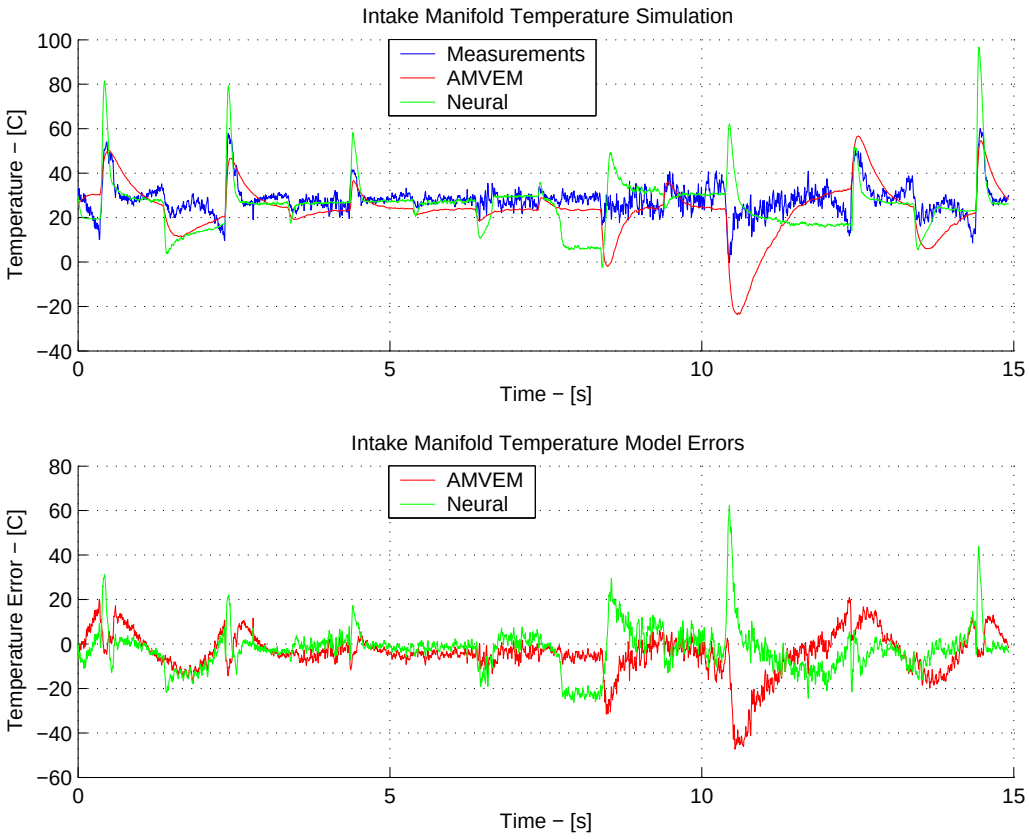


Figure 4.23: Intake Manifold Temperature Simulation - Test Set

4.9 Conclusions

A new β_2 (see equation 4.12) function has been suggested to help the AMVEM model handle intake manifold pressures larger than ambient pressure and the infinitely large gradient of β_2 at a pressure ratio of one. A limited gradient improves the simulation speed of AMVEM models since the integration routines can more easily solve the problem.

A large error in the AMVEM modelling of the intake manifold temperature has been discovered during the work with dynamic neural network models. A plot containing the measured signals from the ECG test engine and the AMVEM simulated signals is presented.

A fallout removal algorithm which can remove most of the fallouts in the measured engine signals has been developed and implemented in MATLAB.

The throttle air mass flow neural network model found in this work was more accurate than the AMVEM version for the test sets similar in structure to the training set, but was equal to or slightly worse in some places on test sets that were not similar in structure to the training set.

The normalized air fuel ratio (lambda) neural network model turned out to be quite difficult to model. There were some peculiar spikes in the lambda signal which the neural network could not be made to model. It was argued that the spikes perhaps was a sensor fault and this might be the reason why the lambda neural network models accuracy was quite bad for some test sets. The lambda model was not successful, but it is believed that it can be made to work if the noise level of the lambda signal can be made smaller.

The intake manifold pressure model was an overall better model than the AMVEM model, but still showed the same kind of reduction in performance on test sets of a different type than the training set. The pressure model is one of the best models achieved in this work which is most like due to the relatively clean pressure signal. This shows the importance of having good sensors when modelling with neural networks.

The intake manifold temperature model was not quite as successful as the intake manifold pressure model. The accuracy on test set with the same structure as the training set is however consistently much better than the AMVEM models temperature output. The missing physics in the AMVEM that causes the large temperature error has been identified by the neural network model. The performance is however reduced on test sets of another type than the training set. The measured temperature signal is however in this work not completely reliable since it is an experimental type of fast temperature sensor. The performance of the neural network temperature model will of course also be affected by this fact.

The experimental fast temperature sensor utilized in this work is not reliable and the accuracy of the sensor is not yet known. The performance of the neural network models trained in this work which utilize the temperature signal (the lambda model, the intake manifold pressure model, the intake manifold temperature model) must thus be viewed in this light. The author of this work believes that the temperature sensor is too noisy at this time and that the high noise level has seriously reduced the performance of the neural network models.

Chapter 5

MIMO Neural Predictive Control

Among the many neural network applications is control. Neural networks can be used for control, both indirectly as system models or as controllers themselves.

Both of these uses are very interesting because neural networks has the potential of becoming a very accurate system model that would otherwise be too complicated to handle with a standard physical approach, or as the almost infinitely flexible controller that can be shaped into giving any control signal necessary.

In this chapter, the first application of a neural network (as a model) is utilized to construct an advanced multi input multi output nonlinear predictive controller (MIMONPC).

5.1 MIMO Nonlinear Predictive Controller

The concept behind the predictive control strategy is described in section 1.4.4 and has been implemented and described for a SISO system by P.M. Nørgaard in [36], [31] and [32].

In this work, a MIMO version of this controller will be developed, described and tested on a SIMULINK MIMO test model.

5.1.1 Algorithm Overview

Predictive control block diagram

A flow chart of the predictive control algorithm can be seen in figure 5.1 as an overview of how the predictive controller algorithm works during one sample period.

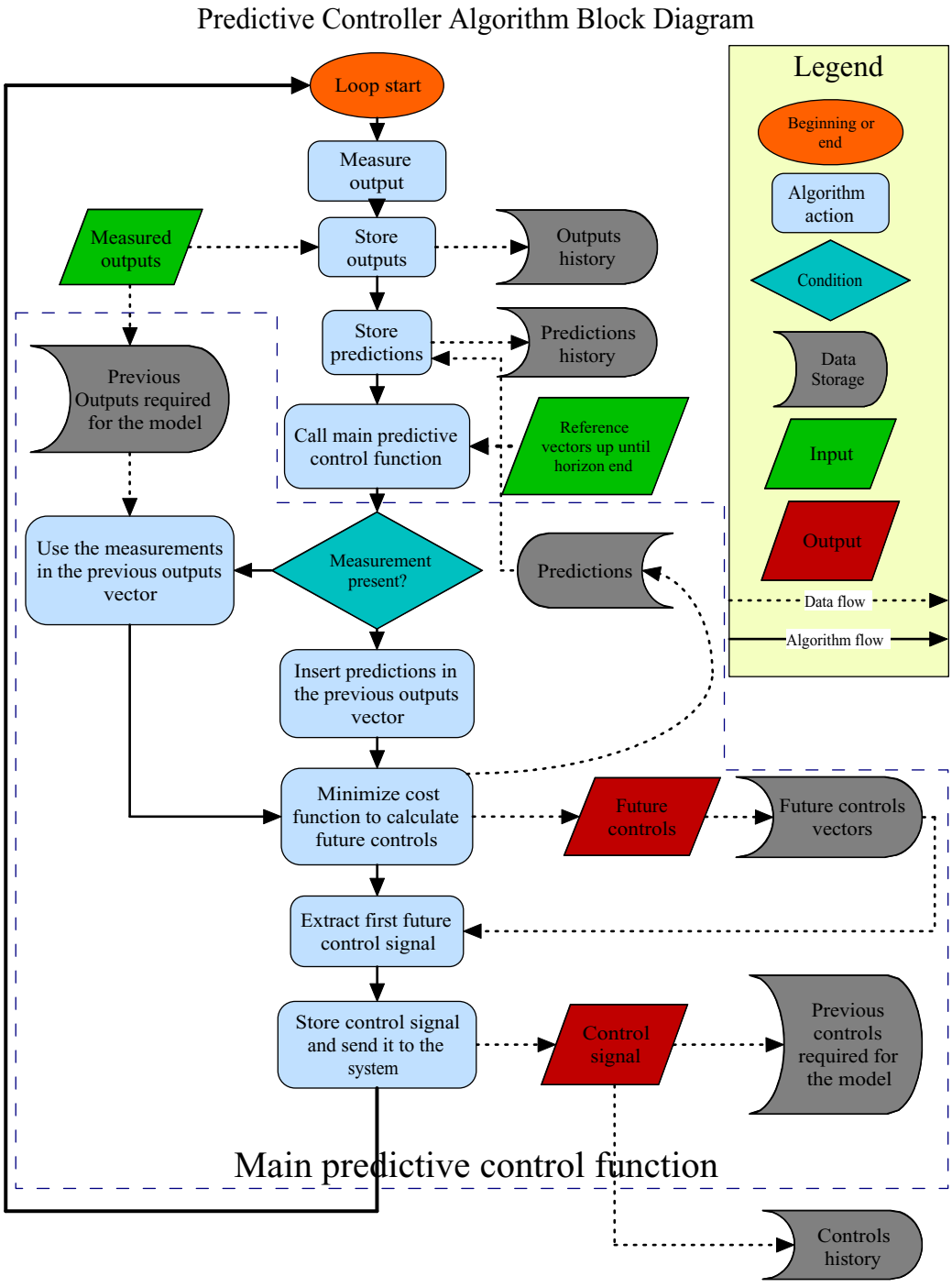


Figure 5.1: Predictive Control Block Diagram

Block diagram description

The following is a list of short descriptions of the steps taken during one sample period in the algorithm.

1. The algorithm begins by obtaining the measured outputs. If there are no measurements available then it is indicated by making an output vector pointer NULL.

2. Store all the outputs and the predictions for later analysis. The first predictions are just made equal to the first measured outputs.

3. Check the output pointer vector. If it is NULL then use a prediction instead, otherwise use the measurement.

4. Minimize the cost function (See 1.7 or later for a more detailed description 5.18) and store all obtained future control vectors as a starting guess for the next minimization.

5. Extract the first of the future control vectors and return it as the control signal.

6. Store the control signal returned for later analysis.

7. Go back to 1.

This is basically how the algorithm works. The following is a more detailed description of the how to make this algorithm work on neural network based MIMO systems.

5.1.2 A Nonlinear MIMO Model

The predictive controller algorithm needs a model to predict the future outputs and thereby the future errors as explained in section 1.4.4.

The model structure chosen for this work is a nonlinear multi input multi output (MIMO) model and later on specialized to be a single hidden layer neural network model. The theory for the controller is, however, at first developed for a general nonlinear MIMO model.

The Model

The nonlinear MIMO model structure used for this purpose will be quite general in order to make it suitable for as many systems as possible. It will look like this.

$$Y_{k+1}^{v} = F(Y_k^{in}, U_k^{in}) \tag{5.1}$$

Where

$$Y_k^{in} = \begin{bmatrix} y_k^1 & y_{k-1}^1 & \cdots & y_{k-n+1}^1 \\ y_k^2 & y_{k-1}^2 & \cdots & y_{k-n+1}^2 \\ \vdots & \vdots & & \vdots \\ y_k^{n_{out}} & y_{k-1}^{n_{out}} & \cdots & y_{k-n+1}^{n_{out}} \end{bmatrix} = \begin{bmatrix} Y_k^v & Y_{k-1}^v & \cdots & Y_{k-n+1}^v \end{bmatrix} \tag{5.2}$$

$$Y_k^v = \begin{bmatrix} y_k^1 & y_k^2 & \cdots & y_k^{n_{out}} \end{bmatrix}^T \tag{5.3}$$

$$U_k^{in} = \begin{bmatrix} u_k^1 & u_{k-1}^1 & \cdots & u_{k-m+1}^1 \\ u_k^2 & u_{k-1}^2 & \cdots & u_{k-m+1}^2 \\ \vdots & \vdots & & \vdots \\ u_k^{n_{in}} & u_{k-1}^{n_{in}} & \cdots & u_{k-m+1}^{n_{in}} \end{bmatrix} = \begin{bmatrix} U_k^v & U_{k-1}^v & \cdots & U_{k-m+1}^v \end{bmatrix} \tag{5.4}$$

$$U_k^v = \begin{bmatrix} u_k^1 & u_k^2 & \cdots & u_k^{n_{in}} \end{bmatrix}^T \tag{5.5}$$

$F()$	is the nonlinear C^1 system model function.
Y_k^{in}	is the system previous outputs matrix at time k.
Y_k^v	is the system output vector at time k.
U_k^{in}	is the system input matrix at time k.
U_k^v	is a system input vector at time k.
y_k^i	is system output i at time k.
u_k^i	is system input i at time k.
n	is the number of output samples back in time needed by the model.
m	is the number of input samples back in time needed by the model.
n_{in}	is the number of inputs.
n_{out}	is the number of outputs.

This model is sufficiently general for most purposes without increasing the complexity to unreasonable levels.

The many input elements and function inputs in this kind of models needs a more compact notation in order to make the presentation simpler. The next section describes a shorthand notation that will be utilized from here on.

Shorthand

The following notation will be used to simplify writing the long series of inputs and outputs and also for function inputs. It is a little like the array range notation in the programming language Pascal.

For function inputs:

$$f(y^i_{a..b}) = f(y^i_a, y^i_{a+1}, \cdots, y^i_b) \tag{5.6}$$

$$f(y^{c..d}_k) = f(y^c_k, y^{c+1}_k, \cdots, y^d_k) \tag{5.7}$$

$$f(y^{c..d}_{a..b}) = f(y^c_a, y^{c+1}_{a+1}, \cdots, y^d_b) \tag{5.8}$$

For matrix rows:

$$[y^i_{a..b}] = \begin{bmatrix} y^i_a & y^i_{a+1} & \cdots & y^i_b \end{bmatrix} \tag{5.9}$$

$$[y^{c..d}_k] = \begin{bmatrix} y^c_k & y^{c+1}_k & \cdots & y^d_k \end{bmatrix} \tag{5.10}$$

$$[y^{c..d}_{a..b}] = \begin{bmatrix} y^c_a & y^{c+1}_{a+1} & \cdots & y^d_b \end{bmatrix} \tag{5.11}$$

For making a lot of terms equal to a certain number:

$$Y_{a..b} = 0 \Leftrightarrow Y_a = Y_{a+1} = \cdots = Y_b = 0 \tag{5.12}$$

following the same pattern as above for the function inputs and the matrix rows.

Note that a could be larger than b implying that the index is decreasing. The same goes for c and d.

Shorthand Example

This notation will turn equation 5.2, 5.3 and 5.4 into.

$$Y^{in}_k = \begin{bmatrix} y^1_{k..k-n+1} \\ y^2_{k..k-n+1} \\ \vdots \\ y^{n_{out}}_{k..k-n+1} \end{bmatrix} \tag{5.13}$$

$$Y_k^v = \left[\; y_k^{1..n_{out}} \; \right]^T \tag{5.14}$$

$$U_k^{in} = \begin{bmatrix} u_{k..k-m+1}^1 \\ u_{k..k-m+1}^2 \\ \vdots \\ u_{k..k-m+1}^{n_{in}} \end{bmatrix} \tag{5.15}$$

$$U_k^v = \left[\; u_k^{1..n_{in}} \; \right]^T \tag{5.16}$$

5.1.3 The Cost Function

Predictive Cost Function

The cost function in a predictive algorithm is a sum of squared errors type of cost function, but it is different from the one utilized in the LQR control algorithm. The difference is that it is future errors rather than past ones that are weighted. This gives the predictive controller the ability to react in "good time".

However, this also means that the reference has to be known for as many sample steps into the future as one wants to minimize the error for. This is not a big problem since the amount of sample steps into the future is usually only in the range of

$$\frac{t_s(5\%)}{T_s} \tag{5.17}$$

See [39]. Where

$t_s(5\%)$	Is the 5% rise time.
T_s	Is the sample time.

The predictive controller algorithm also has the special feature that it minimizes the cost function at each sample time which always makes it find the best possible (limited by the accuracy of the model used and the noise) control signal.

The predictive cost function looks like this in the general case. General in the sense that no time ranges and number of inputs or outputs are specified in this equation.

$$J_k^{npc} = \frac{1}{2}(R_k - \hat{Y}_k)^T(R_k - \hat{Y}_k) + \frac{1}{2}\rho_u \Delta U_k{}^T \Delta U_k \tag{5.18}$$

R_k	Is the future reference vector containing all the future references from the first prediction time to the last.
$\hat{Y}_k$	Is the predicted future outputs vector containing all predictions.
ΔU_k	Is $U_k - U_{k-1}$
U_k	Is the future controls vector containing all future outputs. The result of the minimization process.
ρ_u	Is used to specify how important the changing of the control signal is relative to the control error.

The problem with MIMO is immediately seen in 5.18 since the cost function is more suitable for vectors than matrices.

The model in 5.1 is however designed to take matrices as inputs since it has to handle multiple inputs and multiple outputs thereby making it more practical to keep all the outputs or inputs, belonging to one time step, in the same column as in 5.2 and 5.4.

Notice also that the cost function minimizes the change in the control signal rather than the size of the control signal. This is to prevent the cost function from affecting steady state behavior. Weighing the change in the control signal will not prevent the controller from using large control signals if that is necessary to achieve a zero steady state error.

Special formatting of the signals in the data vectors $R, \hat{Y}$ and U is necessary since MIMO capabilities is desired in this algorithm.

Vector Formats

The reference signal has to be known for some amount of time in advance and it can therefore naturally be put into a matrix as follows.

$$R_k^{matrix} = \begin{bmatrix} r^1_{k+h_s..k+h_e} \\ r^2_{k+h_s..k+h_e} \\ \vdots \\ r^{n_{out}}_{k+h_s..k+h_e} \end{bmatrix} \tag{5.19}$$

Where

r^i_{k+j} Is the reference for output i at time $k+j$
h_s Is how many sample steps into the future from when the error will be calculated, called Horizon Start.
h_e Is how many sample steps into the future to when the error will be calculated, called Horizon End.

The most efficient way to convert this into a vector in a systematic way would be to simply take the rows of 5.19, transpose them and stack them right after each other.

$$R_k = \begin{bmatrix} r^1_{k+h_s..k+h_e} & r^2_{k+h_s..k+h_e} & \cdots & r^2_{k+h_s..k+h_e} \end{bmatrix}^T \tag{5.20}$$

This is the most efficient method since this can be done simply with the Mat2Vec() function (See table A.7).

Mat2Vec() with the second argument taking its default value(true) simply copies the content of the source matrix, changes number of rows to the number of elements in the source matrix and makes the number of columns 1.

This is possible because the matrix elements are stored in a row wise manner.

The data format in 5.20 will be referred to as the Time First Then Number(TFTN) format since increments in time is counted before increments in input/output number.

The "opposite" format will be referred to as Number First Then Time(NFTT) format

The future output vector $\hat{Y}$ has to be formatted in the same way as the future reference vector in 5.20 to make the subtraction in 5.18 meaningful.
Matrix form

$$\hat{Y}^{matrix}_k = \begin{bmatrix} \hat{y}^1_{k+h_s..k+h_e} \\ \hat{y}^2_{k+h_s..k+h_e} \\ \cdots \\ \hat{y}^{h_e-h_s+1}_{k+h_s..k+h_e} \end{bmatrix} \tag{5.21}$$

Vector Form

$$Y_k = \begin{bmatrix} \hat{y}^1_{k+h_s..k+h_e} & \hat{y}^2_{k+h_s..k+h_e} & \cdots & \hat{y}^{h_e-h_s+1}_{k+h_s..k+h_e} \end{bmatrix}^T \tag{5.22}$$

The best format for the future control vector is more easily seen after having taken a closer look at how the prediction of the future outputs are managed and at how

the derivatives of those are calculated.

There is another problem with 5.18 since the Levenberg-Marquardt algorithm is the optimization algorithm chosen for this problem. The cost function consist of two quadratic terms which the standard Levenberg-Margquardt least squares algorithm is not immediately designed to handle.

The next section describes how to modify the Levenberg-Marquardt optimization algorithm making it able to handle multiple quadratic terms.

5.1.4 Optimization Algorithm

The minimization step mentioned in section 5.1.1 is executed at every sample time and should therefore be a relatively simple and efficient optimization algorithm.

The idea is to utilize the previous control signal as the starting guess for the next optimization step in order to minimize the number of optimization iterations necessary to find the next control signal with adequate accuracy.

The Levenberg-Marquardt algorithm is a good choice since it is relatively simple. It only needs the first order derivatives and it is widely used because it has second order convergence properties when close to the optimal point.

These second order convergence properties will almost always be present if the previous control signal is utilized as the initial optimum guess for the next sample step and if the system is assumed to be continuous. Continuity assures that the next control signal will not be far from the previous one if the time between the control signals is short.

The Levenberg-Marquardt algorithm is usually written with only one quadratic term in the function intended for minimization and not with two as in the cost function 5.18. It can, however, easily be extended to multiple quadratic terms.

Multiple Quadratic Term Cost Function.

Consider a cost function of the following form.

$$J(x) = \frac{1}{2}\sum_{i=1}^{N} F^{i^T}(x)F^i(x) \tag{5.23}$$

Where

$$F^i(x) = \left[\; F^i_{1..n} \; \right]^T \tag{5.24}$$

and

n is the number of elements in $F^i(x)$
N is the number of quadratic terms in the cost function.

F^i would typically be a function that returns the error of some signals in interest. For control purposes it would be the error between the reference and the target signals.

Second Order Approximations

The same approximation to the hessian as in the normal Levenberg-Marquardt algorithm can be applied to each of the terms in the sum in 5.23. It can be done as follows.

The first derivative of the cost function 5.23 looks like this.

$$\frac{dJ}{dx} = \sum_{i=1}^{N} \left(\frac{dF^i(x)}{dx} \right)^T F^i(x) \tag{5.25}$$

Where

$$\frac{dF^i(x)}{dx} = \begin{bmatrix} \frac{dF^i_1}{dx_{1..m}} \\ \vdots \\ \frac{dF^i_n}{dx_{1..m}} \end{bmatrix} \tag{5.26}$$

and m is the number of elements in x

The second derivative of the cost function 5.23 looks like this.

$$\frac{d^2J}{dx^2} = \sum_{i=1}^{N} \left(\left(\frac{dF^i(x)}{dx} \right)^T \frac{dF^i(x)}{dx} + \left(\frac{d^2F^i(x)}{dx^2} \right)^T F^i(x) \right) \tag{5.27}$$

The approximation is then to simplify the second derivative term in equation 5.27 since it would be most difficult to calculate and would put an heavy load on the CPU in real-time applications. It is also unnecessary to calculate this term accurately since a sufficient approximation to this term can be made.

Each term belonging to the same sum index i in equation 5.27 is approximated by replacing the second derivative term with

$$\left(\frac{d^2F^i(x)}{dx^2}\right)^T F^i(x) \approx \lambda I \tag{5.28}$$

Where

- λ is a constant.
- I is a $m \times m$ unity matrix.

and thus equation 5.27 becomes

$$\frac{d^2J}{dx^2} = \sum_{i=1}^{N}\left(\left(\frac{dF^i(x)}{dx}\right)^T\left(\frac{dF^i(x)}{dx}\right) + \lambda I\right) \tag{5.29}$$

The approximation method explained here is the one used in the Marquardt optimizer class (See section A.6) for more about the Marquardt C++ class.

Minimization Step

The optimization algorithm works in all other ways like the regular Levenberg-Marquardt algorithm. It makes a second order approximation of the cost function 5.18 at the current optimum guess, called x_0 here.

$$J(x_0 + \Delta x) \approx J(x_0) + \frac{dJ}{dx}(x_0)\Delta x + \frac{1}{2}\Delta x^T\frac{d^2J}{dx^2}(x_0)\Delta x \tag{5.30}$$

The intention is then to find the Δx that yields the smallest value of the second order approximation by solving.

$$\begin{array}{lcl} \frac{dJ}{d\Delta x}(x_0 + \Delta x) & = & 0 \\ \Updownarrow & & \\ \frac{dJ}{dx}(x_0) + \frac{d^2J}{dx^2}(x_0)\Delta x & = & 0 \\ \Updownarrow & & \\ \Delta x & = & -\left(\frac{d^2J}{dx^2}(x_0)\right)^{-1}\frac{dJ}{dx}(x_0) \end{array} \tag{5.31}$$

The Levenberg-Marquardt algorithm then adds Δx to the current optimum guess x_0 if it will result in a reduction of the cost function 5.18 and calculates the derivatives again in the new point.

If the step Δx does not result in a reduction or too small a reduction of the value of the cost function 5.18 then an increase of λ will take place and the step length

will thus be reduced. Furthermore, the step taken is thus also closer to a steepest descent step when λ is increased since

$$\frac{d^2J}{dx^2} = \sum_{i=1}^{N} \left(\left(\frac{dF^i(x)}{dx} \right)^T \left(\frac{dF^i(x)}{dx} \right) + \lambda I \right) \approx \lambda I \tag{5.32}$$

when λ is large and Δx then becomes

$$\Delta x = -\left(\frac{d^2J}{dx^2}(x_0) \right)^{-1} \frac{dJ}{dx}(x_0) \approx -\frac{1}{\lambda} I \frac{dJ}{dx}(x_0) = -\frac{1}{\lambda}\frac{dJ}{dx}(x_0) \tag{5.33}$$

which is a dampened steepest descent step.

If the step instead results in a large reduction of the value of the cost function then a decrease of λ will take place and the step length will be increased. The step will also be closer to a Gauss- Newton step assuming that the hessian approximation was good.

A short step by step description of the algorithm starting at the optimum guess x_n is shown below.

1. $n = 0$
2. Calculate Δx using 5.31
3. $cost = J(x_n)$ (see 5.18)
4. $x_{new} = x_n + \Delta x$
5. $newcost = J(x_{new})$
6. Update λ
7. $n = n + 1$
8. If $newcost < cost$ then $x_n = x_{new}$ else $x(n) = x(n-1)$
9. If $\left(\frac{dJ}{dx}(x_n)\right)^T \frac{dJ}{dx}(x_n) > StopCriterion$ then goto 2

The λ is updated at each optimization iteration based on how good an approximation 5.30 is of 5.23. It is updated according to the scheme explained in [19] in the following way.

$$\lambda_{n+1} = \begin{cases} 2\lambda_n & if \frac{\text{Actual decrease}}{\text{Estimated decrease}} < \frac{1}{4} \\ \frac{1}{2}\lambda_n & if \frac{\text{Actual decrease}}{\text{Estimated decrease}} > \frac{3}{4} \end{cases} \tag{5.34}$$

Where

$$\text{Actual decrease} = cost - newcost \tag{5.35}$$

The estimated decrease is found by inserting Δx from 5.31 in 5.30.

$$\begin{aligned} \text{Estimated decrease} &= -\Delta J(x_0) = -\left(J(x_0+\Delta x) - J(x_0)\right) \\ &= \tfrac{1}{2}\left(\tfrac{dJ}{dx}(x_0)\right)^T \left(\tfrac{d^2J}{dx^2}(x_0)\right)^{-1} \tfrac{dJ}{dx}(x_0) \\ &= (\Delta x)^T \tfrac{dJ}{dx}(x_0) \end{aligned} \tag{5.36}$$

Note on Multiple Lambdas

An attempt to use individual lambda's for each term has been made, but was found to only make things more complicated without also improving the speed of the optimization(Reducing the number of iterations necessary to reach the optimum within the specified accuracy).

5.1.5 Prediction Derivatives

The term "predictions" will in the following refer to values of $\hat{Y}_{k+i}$ for $i \geq 1$.

The previous section shows that the Levenberg-Marquardt optimizing algorithm needs the derivative of both the terms in the cost function 5.18 being squared. Those terms are

$$E_k = (R_k - \hat{Y}_k) \tag{5.37}$$

and

$$\sqrt{\rho_u}\Delta U_k \tag{5.38}$$

This is seen by comparing 5.18 with 5.23.

F_1 in 5.23 corresponds to 5.63 and F_2 in 5.23 corresponds to 5.64.

The most complicated one being the derivative of the predicted errors 5.37. This implies that the derivative of the predicted output is needed.

$$\frac{\partial E_k}{\partial U_k} = -\frac{\partial \hat{Y}_k}{\partial U_k} \tag{5.39}$$

since R_k is considered constant.

The following sections explains how to calculate the derivatives of the squared terms in the predictive control cost function beginning with the predicted error term 5.37.

Iterative Prediction

The values in the future output vector 5.22 are predicted by using the model of the system iteratively in the following fashion.

$$\begin{array}{lcl} \hat{Y}^v_{k+1} & = & F(Y^{in}_k, U^{in}_k) \\ \hat{Y}^v_{k+2} & = & F(\hat{Y}^{in}_{k+1}, U^{in}_{k+1}) \\ \cdots\cdots & & \end{array} \tag{5.40}$$

Where $\hat{Y}^{in}_{k+i}$ contains predictions for future values of Y^{in} and/or measurements for current or past values of Y^{in}.

This continues for as many times as needed which is until $\hat{Y}^v_{k+h_e}$ has been calculated.

The output of the the model function is a vector and is most easily stored in a matrix by concatenating the vectors with the | operator from the matrix library (See table A.3) as they are calculated.

The last step in that process is illustrated more clearly below. The vertical line represents the concatenation.

$$\hat{Y}^{matrix}_k = \left[\begin{array}{c|c} \hat{y}^1_{k+h_s..k+h_e-1} & \hat{y}^1_{k+h_e} \\ \hat{y}^2_{k+h_s..k+h_e-1} & \hat{y}^2_{k+h_e} \\ \vdots & \vdots \\ \hat{y}^{n_{out}}_{k+h_s..k+h_e-1} & \hat{y}^{n_{out}}_{k+h_e} \end{array}\right] \tag{5.41}$$

This matrix is then easily changed into the future outputs vector 5.22 by the Mat2Vec() function.

Future Control Signal Time Limit

The series of future control vectors $U^v_k, U^v_{k+1}, \cdots$ in 5.40 are the vectors to be found by optimization and the derivative of the cost function is to be taken with

respect to those vectors as explained previously in the optimization section 5.1.4.

The second order hessian approximation 5.29 for the predicted error term 5.37 is of particular concern since it will grow in size quadratically with the number of future controls necessary to calculate the last predicted output.

The complete optimization step including the hessian in 5.31 applied on 5.18 looks like this.

$$\Delta x_k = -\left(\frac{\partial^2 J_k^{npc}}{\partial U_k^2}\right)^{-1} \frac{\partial J_k^{npc}}{\partial U_k} \tag{5.42}$$

The inverse of the hessian is taken in 5.42 and will therefore quickly become a large computational burden with an increased number of future controls.

This problem is normally handled by limiting the number of future controls that is free for the optimizing algorithm to change. If all the future controls needed to calculate $\hat{Y}^v_{k+h_e}$ were variable then the hessian would be an $h_e \times h_e$ matrix.

Instead a limit, $h_c < h_e$, on the number of free future controls is forced. h_c is the last control (U_{k+h_c}) that is considered free for the optimizer to change. The rest of the future controls needed to calculate $\hat{Y}^v_{k+h_e}$ which is $(h_e - h_c - 1)$ will be held constant at the last free value (U_{k+h_c}).

The last values in the future control vector that are held constant are therefore not part of the vector U in 5.18 but is only used in its entire length when calculating the predictions for times later than $\hat{Y}_{k+h_c+1}$.

Two things about the future control vector still remains to be determined.

1. Should the NFTT format be used or the TFTN format?

2. Should time be counted up or down with increasing column numbers?

The answer to those two questions depends strongly on how the derivatives in 5.42 are calculated.

The next sections describes how to calculate the derivatives in an efficient way which is important because the CPU load should be kept as low as possible.

Efficiency

The Levenberg-Marquardt optimization method requires the derivative of the predictive error function (called F_i in section 5.1.4) with respect to the control vector. This is the term 5.37.

It is necessary to rewrite the model function 5.1 in order to find a method to calculate this derivative in an efficient way.

The derivative will be calculated analytically. It is important to make the calculation of the derivative analytic since a numerical calculation would have to call the iterative prediction procedure for as many times as there are variable controls in the future control vector.

Furthermore, the iterative predictive procedure calls the model function for as many times as specified by the difference between the horizon end and the horizon start parameter (h_e and h_s, see 5.19).

The analytic form of the derivative will significantly reduce the amount of calculations necessary and also reduce the amount of numerical noise. The analytical form is also necessary in order to obtain an efficient iterative formula which will reduce the amount of calculations even more since many of the terms can be reused. This will be easier to see in the following.

The matrix input and output form of the model in 5.1 is however not quite suitable for finding a method to calculate an analytical derivative.

To help with this calculation, the model functions two inputs will be split into as many vectors as they have columns so that each column represent one time step.

Splitting Up the Inputs of the Model Function

Splitting up the inputs to the model in 5.1 makes it look like this.

$$\hat{Y}^v_{k+1} = F(Y^v_{k..k-n+1}, U^v_{k..k-m+1}) \tag{5.43}$$

Where

Y^v_{k-j} is an output vector at time $k-j$.
U^v_{k-j} is an input vector at time $k-j$.

The original model function structure with matrix inputs and outputs (5.1) will still be used externally (as seen from the user of the predictive controller algo-

rithm).

The split version (5.43) is just a temporary form utilized in the process of finding an algorithm capable of calculating the derivative of the predictive cost function 5.18.

Predictor Input Structure

It is useful to take a look at what the prediction equation structure looks like before calculating the derivatives.

The predictor/model input structure varies a little depending on the how large h_e(see 5.1.3) is compared to n(see 5.1.3).

If $n < h_e$ then all the past outputs will be predictions from the prediction of Y^v_{k+n+1} to the prediction of $Y^v_{k+h_e}$.

If $n \geq h_e$ then not all the past outputs will be predictions, but actual measurements since the number of necessary previous outputs then exceeds the number of predictions necessary to produce the last prediction $Y^v_{k+h_e}$.

Special care has to be taken with the control signals since they will become constant after time $k + h_c$. The control signal vectors U^v_{k+j} will be written as $U^v_{k+h_c}$ after time $k + h_c$ to show that the control signals are held constant.

The predictor/model input structure also depends on the number of previous inputs m compared to the difference $h_e - h_c$.

If $m \leq h_e - h_c$ then all the m previous inputs used to calculate each of the predictions from $Y^v_{k+h_c+m}$ to $Y^v_{k+h_e}$ will be equal to the last free control signal $U^v_{k+h_c}$.

If $m > h_e - h_c$ then only some of the previous inputs used to calculate each of the predictions from Y^v_{k+ch+m} to $Y^v_{k+h_e}$ will be equal to the last free control signal $U^v_{k+h_c}$ and the rest will continue to be equal to the free future control signals.

The following equations illustrates how this works for the case:

$n \geq h_e \quad , \quad m > h_e - h_c$.

$$
\begin{array}{lcl}
\hat{Y}^v_{k+1} & = & F(Y^v_{k..k-n+1}, U^v_{k..k-m+1}) \\
\hat{Y}^v_{k+2} & = & F(\hat{Y}^v_{k+1}, Y^v_{k..k-n+2}, U^v_{k+1..k-m+2}) \\
\vdots & & \vdots \\
\hat{Y}^v_{k+h_c+1} & = & F(\hat{Y}^v_{k+h_c..k+1}, Y^v_{k..k-n+h_c}, U^v_{k+h_c..k-m+h_c+1}) \\
\hat{Y}^v_{k+h_c+2} & = & F(\hat{Y}^v_{k+h_c+1..k+1}, Y^v_{k..k-n+h_c+1}, U^v_{k+h_c}, U^v_{k+h_c..k-m+h_c+2}) \\
\vdots & & \vdots \\
\hat{Y}^v_{k+h_e} & = & F(\hat{Y}^v_{k+h_e-1..k+1}, Y^v_{k..k-n+h_e}, \underbrace{U^v_{k+h_c}, \cdots, U^v_{k+h_c}}_{h_e-h_c}, U^v_{k+h_c-1..k-m+h_e})
\end{array}
\tag{5.44}
$$

Calculating the Individual Derivatives

Derivatives have to be found for all future outputs with respect to all future controls. This is most easily done at first by calculating the derivatives individually. The first future output with respect to the first future control, then the first future output with respect to the second future control and so on.

The future controls $U^v_{k+h_c..k+h_e-1}$ that are held constant and equal to $U^v_{k+h_c}$ are not taken into account in the first place to make the analysis simpler. That special case is taken care of later on.

The following notation will be used for the partial derivatives of the model function F (See 5.43).

$$\dot{F}^i_j = \frac{\partial F}{\partial x_j} \tag{5.45}$$

meaning the partial derivative of F with respect to input number j and evaluated at the inputs utilized for the prediction of $\hat{Y}^v_{k+i}$. For instance.

$$\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_k} = \dot{F}^1_{n+1} = \frac{\partial F}{\partial x_{n+1}}(Y^v_{k..k-n+1}, U^v_{k..k-m+1}) \tag{5.46}$$

x_{n+1} refers to the $(n+1)th$ argument of the function F in 5.43.

The inputs used for $\dot{F}^1_{n+1}()$ in the above case are the ones from line 1 in equation 5.44.

The derivatives for the first three predictions are calculated and displayed in the following in order to make a pattern in the derivatives clearer.

The first derivatives can be easily obtained by using the first line in the prediction equations in 5.44.

$$\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_k} = \dot{F}^1_{n+1} \tag{5.47}$$

The rest of the derivatives for the first prediction are merely zero vectors since no future controls other than U^v_k exist in the equation (first line in 5.44) for the first prediction($\hat{Y}^v_{k+1}$).

$$\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_{k+1..k+h_c}} = 0 \tag{5.48}$$

The derivatives for the second prediction ($\hat{Y}^v_{k+2}$) can be obtained by using the second line in the prediction equations 5.44.

The chain rule for differentiation will have to be applied for the first derivative ($\frac{\partial \hat{Y}^v_{k+2}}{dU^v_k}$) since the model function will then contain the previous prediction ($\hat{Y}^v_{k+1}$).

$$\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_k} = \dot{F}^2_1 \left(\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_k} \right) + \dot{F}^2_{n+2} \tag{5.49}$$

and more easily for U^v_{k+1} since no other inputs to the model function for $\hat{Y}^v_{k+2}$ depends on U^v_{k+1} but U^v_{k+1} itself.

$$\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_{k+1}} = \dot{F}^2_{n+1} \tag{5.50}$$

The rest of the derivatives for $\hat{Y}^v_{k+2}$ are merely zero because no future controls from time $k+2$ and up are present in the prediction equations (5.44).

$$\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_{k+2..k+h_c}} = 0 \tag{5.51}$$

The derivatives of the third prediction($\hat{Y}^v_{k+3}$) are calculated in the same manner, using the chain rule, and are.

$$\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_k} = \dot{F}^3_1 \left(\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_k} \right) + \dot{F}^3_2 \left(\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_k} \right) + \dot{F}^3_{n+3} \tag{5.52}$$

$$\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_{k+1}} = \dot{F}^3_1 \left(\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_{k+1}} \right) + \dot{F}^3_{n+2} \tag{5.53}$$

$$\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_{k+2}} = \dot{F}^3_{n+1} \tag{5.54}$$

$$\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_{k+3..k+h_c}} = 0 \tag{5.55}$$

Notice here how the expression for the derivative $\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_k}$ (5.52) contains the derivatives $\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_k}$ and $\frac{\partial \hat{Y}^v_{k+1}}{\partial U^v_k}$. This can be taken advantage of in an iterative algorithm that reutilizes already calculated derivatives. The various partial derivatives just have to be calculated in the appropriate order.

In the calculations of the derivatives above it is assumed that n is at least 2 and that m is at least 3 since the chain rule can of course not progress down through inputs to the model function that are not there.

If for instance $n = 1$ and $m = 1$ then 5.52 would look like this.

$$\frac{\partial \hat{Y}^v_{k+3}}{\partial U^v_k} = \dot{F}^3_1 \left(\frac{\partial \hat{Y}^v_{k+2}}{\partial U^v_k} \right) \tag{5.56}$$

since U^v_k and $\hat{Y}^v_{k+1}$ would not be present in the prediction equation in this case.

Iterative Prediction Derivative Calculation

The calculations of the derivatives of the third predictions in 5.52, 5.53, 5.54 and 5.55 reveals a pattern in all the derivatives of all predictions.

These four equations and the derivatives of other predictions not shown in this chapter confirms that the derivatives can in general be written as the following sum.

$$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}} = \begin{cases} 0 & , i \leq j \\ \sum_{r=1}^{N_t} \dot{F}^i_{x_r} \frac{\partial \hat{Y}^v_{k+i-r}}{\partial U^v_{k+j}} + \begin{cases} 0 & , i-j > m \\ \dot{F}^i_{x_{n+i-j}} & , i-j \leq m \end{cases} & , i > j \end{cases} \tag{5.57}$$

$$N_t = min(n, i-j-1) \tag{5.58}$$

N_t is the number of terms in the sum and clearly depends on the difference between i and j as shown in 5.58.

The sum comes from the use of the chain rule as it progresses down through the inputs of the model function $F()$, but it cannot progress any further than there are previous outputs (n) in the model function. This is the reason for the $min(x, y)$ function (returns the smallest of the two inputs) and n is the number of previous outputs given as inputs to the model function.

$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}}$ is of course 0 when $i \leq j$ since none of the inputs to the model function will then depend on the future control signal vector that the derivative is taken with respect to. This is because U^v_{k+j} only can affect the system output for times later than $k + j$.

The last term added after the sum, 0 or $\dot{F}^i_{x_{n+i-j+1}}$, comes from the previous inputs part of the system models inputs. 0 if the input is not present because the prediction is to far out in the future compared to the number of previous inputs to the system. Otherwise the derivative of the system model function with respect to the inputs where U_{k+j} appears is taken and added as a direct consequence of the chain rule.

Constant Future Controls

The formula for the derivatives $\frac{\partial Y^v_{k+i}}{\partial U^v_{k+j}}$ in 5.57 are correct for all values of j except for $j = h_c$ if the future controls $U^v_{k+h_c..k+h_e-1}$, being held constant and equal to $U^v_{k+h_c}$, are taken into account.

This is because the control signal $U^v_{k+h_c}$ is repeated for times larger than $k + h_c$ and the term behind the second soft bracket in equation 5.57 is not the only one coming from using the chain rule on the model function when deriving with respect to $U^v_{k+h_c}$.

The term behind the second soft bracket in equation 5.57 should then be replaced by another condition where $j < h_c$ equals the previous term and $j = h_c$ equals a sum. A sum of maximum m elements since the chain rule can not progress down through more inputs than the model function takes.

The number of terms is not always m since the control signal is only repeated for times larger than $k + h_c$. The number of terms must be equal to $i - h_c$ since that is

the number of time steps from time $k + h_c$ to time $k + i - 1$ which is the number of control signals necessary to produce the future output Y^v_{k+i}.

The complete function for the derivative of Y^v_{k+i} with respect to U^v_{k+j} is then

$$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}} = \begin{cases} 0 & , i \leq j \\ \sum_{r=1}^{N_t^y} \dot{F}^i_{x_r} \frac{\partial \hat{Y}^v_{k+i-r}}{\partial U^v_{k+j}} + \begin{cases} 0 & , i-j > m \\ \dot{F}^i_{x_{n+i-j}} & , i-j \leq m \end{cases} & , j < h_c \\ \sum_{q=1}^{N_t^u} \dot{F}^i_{x_{n+q}} & , j = h_c \\ & , i > j \end{cases} \tag{5.59}$$

$$N_t^y = min(n, i-j-1) \quad , \quad N_t^u = min(m, i-j) \tag{5.60}$$

Iterative Derivative Algorithm - Overview

The results in the previous section shows that an iterative algorithm can be constructed in order to calculate the derivative of the predicted outputs in an efficient way. Efficiently by reutilizing previously calculated derivatives.

Reutilization of previously calculated derivatives require that the partial derivatives

$$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}} \quad , \quad 1 \leq i \leq h_e \text{ and } 0 \leq j \leq h_c \tag{5.61}$$

are calculated in the right order.

This is where the order of the elements in the future controls vector U_k in the cost function 5.18 becomes important.

The order of those elements has to be constructed in way so that not only reutilization of previously calculated partial derivatives 5.61 becomes possible, but also makes it possible to construct the full error derivative matrix

$$\frac{d\hat{E}_k}{dU_k} \tag{5.62}$$

in a practical way.

The optimizing algorithm utilized by the predictive controller will need the derivative of both terms in the cost function 5.18 being squared as can be seen in equation 5.25.

Those terms are.

$$E_k = (R_k - \hat{Y}_k) \tag{5.63}$$

and

$$\sqrt{\rho_u}\Delta U_k \tag{5.64}$$

The complete derivative of 5.63 with respect to the future controls vector is a large matrix formatted according to the formats of $\hat{Y}_k$,R_k and U_k.

The most practical format for the U_k vector will then be based on the most practical format of the derivative of the term E_k.

A practical format (with respect to an efficient iterative algorithm) for that derivative would be this format

$$\frac{d\hat{E}_k}{dU_k} = -\begin{bmatrix} \frac{\partial \hat{Y}^v_{k+h_s}}{\partial U^v_{k..k+h_c}} \\ \frac{\partial \hat{Y}^v_{k+h_s+1}}{\partial U^v_{k..k+h_c}} \\ \vdots \\ \frac{\partial \hat{Y}^v_{k+h_e}}{\partial U^v_{k..k+h_c}} \end{bmatrix} \tag{5.65}$$

since this form makes it easy to construct a set of loops that calculate the individual partial derivatives

$$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}} \quad , \quad 1 \leq i \leq h_e \text{ and } 0 \leq j \leq h_c \tag{5.66}$$

utilizing equation 5.59 one by one for all relevant values of i and j.

The iterative algorithm constructing the predicted error derivative 5.65 starts with $i = 1, j = 0$ and progresses through $j = 0$ to $j = h_c$, increases i by one and so on.

This ensures that derivatives of previous predictions have already been calculated when they are needed for the calculation of later predictions.

The chosen format for the predicted error derivative matrix 5.65 will also make it practical to fill the predicted error derivative 5.65 with elements row by row in an efficient manner.

There will be a C++ program listing in the pseudo code program section 5.1.7 showing how to construct the predicted error derivative 5.65. The entire predictive control C++ code will also be on the CD-ROM that comes with this dissertation.

The Future Control Vector Format

The chosen format for the error derivative shows that U_k should be in then NFTT format since each individual partial derivative 5.61 in the predicted error derivative 5.65 has the following structure.

$$\frac{\partial \hat{Y}^v_{k+i}}{\partial U^v_{k+j}} = \begin{bmatrix} \frac{\partial \hat{y}^1_{k+i}}{\partial u^{1..n_{in}}_{k+j}} \\ \frac{\partial \hat{y}^2_{k+i}}{\partial u^{1..n_{in}}_{k+j}} \\ \vdots \\ \frac{\partial \hat{y}^{n_{out}}_{k+i}}{\partial u^{1..n_{in}}_{k+j}} \end{bmatrix} \tag{5.67}$$

The number of the input is counted in each row in 5.67 and the time indices are constant. Each row in 5.67 will appear in a column block in 5.65 where the time index in each block is j and constant. This is the definition of the NFTT format.

The NFTT format form of the future control vector U_k looks like this.

Matrix form

$$U^{matrix}_k = \begin{bmatrix} u^{1..n_{in}}_k \\ u^{1..n_{in}}_{k+1} \\ \cdots \\ u^{1..n_{in}}_{k+h_c} \end{bmatrix} = \begin{bmatrix} (U^v_k)^T \\ (U^v_{k+1})^T \\ \cdots \\ (U^v_{k+h_c})^T \end{bmatrix} \tag{5.68}$$

and in vector form by utilizing the $Mat2Vec()$ function as with the future reference vector in 5.20

$$U_k = \begin{bmatrix} u^{1..n_{in}}_k & u^{1..n_{in}}_{k+1} & \cdots & u^{1..n_{in}}_{k+h_c} \end{bmatrix}^T \tag{5.69}$$

The Control Cost Derivative

The remaining derivative to be found is the derivative of the control cost term.

$$J_k^U = \sqrt{\rho_u}\Delta U_k \tag{5.70}$$

Displaying the contents of ΔU_k and U_k will make it easier to see what the control cost derivative is.

$$U_k = \begin{bmatrix} u_k^1 \\ u_k^2 \\ \vdots \\ u_k^{n_{in}} \\ u_{k+1}^1 \\ u_{k+1}^2 \\ \vdots \\ u_{k+ch}^{n_{in}} \end{bmatrix} \quad , \quad \Delta U_k = U_k - U_{k-1} = \begin{bmatrix} u_k^1 & - & u_{k-1}^1 \\ u_k^2 & - & u_{k-1}^2 \\ & \vdots & \\ u_k^{n_{in}} & - & u_{k-1}^{n_{in}} \\ u_{k+1}^1 & - & u_k^1 \\ u_{k+1}^2 & - & u_k^2 \\ & \vdots & \\ u_{k+ch}^{n_{in}} & - & u_{k+ch-1}^{n_{in}} \end{bmatrix} \tag{5.71}$$

The control cost derivative is then simply

$$\frac{\partial J_k^U}{\partial U_k} = \begin{bmatrix} I & 0 & \cdots\cdots\cdots & 0 \\ -I & I & \ddots & \vdots \\ 0 & -I & \ddots \quad \ddots & \vdots \\ \vdots & 0 & \ddots \quad \ddots \quad \ddots & \vdots \\ \vdots & \vdots & \ddots \quad \ddots \quad \ddots & 0 \\ 0 & 0 & \cdots \quad 0 \quad -I & I \end{bmatrix} \tag{5.72}$$

Where I is the unity matrix.

5.1.6 System Model Derivatives

The prediction derivatives pattern has been found and written in an iteratively friendly form as a sum of products of the derivatives of the system model function and earlier predictions.

The system model chosen for this work is naturally the single hidden layer neural network since neural network engine control is the topic of this dissertation and because the single hidden neural network is mathematically simple neural network structure that allows for fast and analytic derivative functions.

The system model functions are thus already found in the section 3.5.

5.1.7 Pseudo Code For the Controller

Simplified pseudo code programs written in a C like style for the neural network based nonlinear predictive controller are presented in this section. This is done in order to gather all the pieces of the controller developed in the previous sections and to explain in greater detail their place in the controller algorithm.

A C++ class implementation of the predictive controller and a neural predictive controller has been developed and the source code for them can be found in the Matrix Control Library/Controllers folder on the source code appendix CD coming with this dissertation.

The main controller sub routine has the following structure.

```
Controller()
{
    InitializeStorageArrays();  // For data history.
    CreateControllerObject();   // Instantiate the NPC, the
                                // NeuralPredictive-
                                // Controller class.
    CreateNeuralNetwork();      // Instantiate the Neural
                                // Network class. See A.14.
    InitializeOptimizer();      // Stop Norm etc.
    InitializeModel();          // Initialize all previous
                                // inputs and outputs.
    for(i=1,i<NumberOfControlSteps,i++)
    {
        StoreOutputs();         // Store outputs in a
                                // vector for later
                                // analysis.
        StorePredictions();     // Store predictions in a
                                // vector for later
                                // analysis.
        GetReference();         // Obtain the reference for
                                // h_e steps into the
                                // future.
                                // See section 5.1.3.
        Control=PredControl();  // Calculate the predictive
                                // Control output.
        StoreControlSignal();   // Store control signal
```

```
                                // for later analysis.
        Send(Control);          // Apply the predictive
                                // control signal.
        WaitForNextSampleTime();
        ReadOutputs();          // Read the system outputs.
    }
}

Matrix PredControl()
// This is basically the Control() function in the
// PredictiveControl class.
{
    FutureU=Minimize(FutureU)   // Call the Levenberg-
                                // Marquardt optimizer with
                                // the previous future
                                // control signals as the
                                // initial guess. See
                                // section A.6 for more
                                // about the Marquardt
                                // class. See section 5.1.4
                                // for more about the
                                // Marquardt algorithm.
                                // The cost function 5.18
                                // is minimized by this
                                // call. FutureU is 5.69
    Control=Extract(FutureU);   // Get U_k^v
                                // The first row in 5.68.
    return Control;             // Return the control
                                // signal.
}

Matrix Minimize(Matrix X)
// This function is a part of the Marquardt class
// See section A.6.
{
    Cost=CalculateCost();       // Calculate the value of
                                // the cost function 5.18.
                                // The predictions needed
                                // for this calculation are
                                // found as described
                                // in section 5.1.5.
    CalcGradientHess();         // Calculates the gradient
                                // of the cost function
                                // 5.18 with respect
```

```
                                // to the future controls.
                                // Also calculates the
                                // approximated hessian
                                // 5.29.
    // Now iterate until the gradient is as
    // close to zero norm as specified by StopNorm.
    While(Norm(Gradient)>StopNorm)
    {
        NewStep=
        Inv(Hessian)*Gradient;  // Calculate new step.
        NewCost=
        CalcNewCost(NewStep);   // Calculate the cost at
                                // new point using 5.18
                                // and the method in
                                // section 5.1.5.
        AdjustLambda();         // Adjust the λ value
                                // as explained in 5.34.
        if(NewCost<Cost)        // Useful Step?
        {
            X=X+Step;           // Accept step.
            CalcGradientHess(); // Calculate the new
                                // gradient and hessian.
        }
        else
        {
            AdjustHessian();    // Adjust the hessian
                                // with the new λ
                                // value.
        }
    }
}

CalcGradientHess()
{
    CalcPredJacobian();         // Calculate the derivative
                                // of the terms in the.
                                // cost function 5.18.
                                // The derivatives 5.59
                                // and 5.72.
    E           = PredError();  // Calculate the predictive
                                // error vector =
                                // E from equation 5.37.
    JacobianT   = Transpose(Jacobian);
    Hessian     = Hessian + JacobianT * Jacobian;
```

```
    Gradient    = Gradient + JacobianT * E;
    Cost        = Cost + SqrSum(E);
}
```

5.2 Demonstration

This section contains a demonstration of the MIMO nonlinear predictive controller (MIMO NPC) on a nonlinear test model created in SIMULINK. The MIMO NPC performance will be compared to a discrete linear H_2 controller designed on a linearized version of the nonlinear test model.

All the controllers in this demonstration and comparison experiment are discrete and a sampling time of

$$T = 0.1s \tag{5.73}$$

will be used for all controllers and system models in this demonstration section.

5.2.1 The Test System

A test system with 2 inputs and 2 outputs will be chosen for a demonstration of the MIMO features in the MIMO NPC developed in this chapter. The test system equations are as follows:

$$\begin{aligned} \frac{d^2y_1}{dt^2} &= -\frac{dy_1}{dt} + y_2 - y_1 - y_1^3 + u_1 \\ \frac{d^2y_2}{dt^2} &= -\frac{dy_2}{dt} + y_1 - y_2 - y_2^3 + u_2 \end{aligned} \tag{5.74}$$

A SIMULINK block diagram of the test system is shown in figure 5.2.

5.2.2 Test Model Neural Network Training

Training Data Generation

The data for the network training is obtained by sending adequately exciting input signals through the test system and record the test systems output data. The input and output data from the test model is utilized as learning material for the neural network.

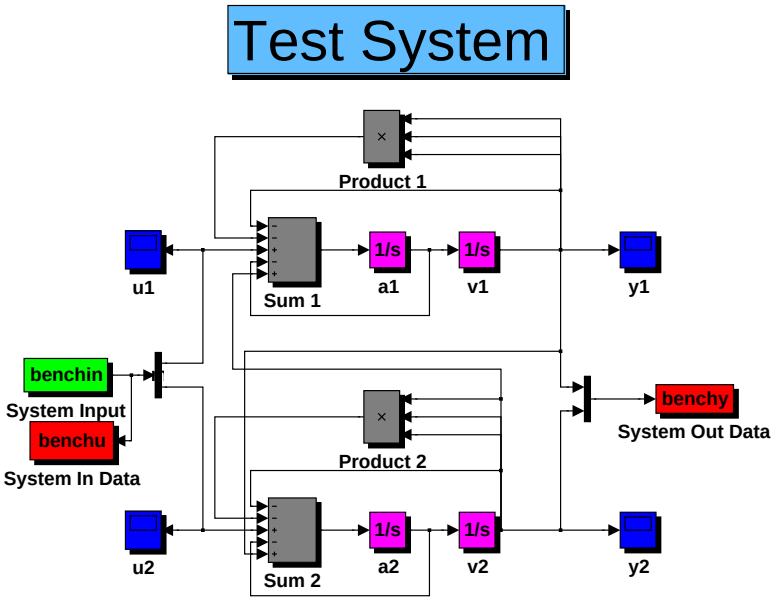


Figure 5.2: MIMO NPC Test Model

The adequately exciting input signals is generated in a random manner. The input signals utilized in this demonstration can be seen in figure 5.3.

The input signals in figure 5.3 are utilized as inputs to the Simulink model shown in figure 5.2 and the resulting output signals are stored. The output signals can be seen in figure 5.4.

Neural Network Training

The neural network is trained with the parameters shown in table 5.1 by the predictive neural network training method developed in chapter 3. These parameters were found to be the best choice with respect to the lowest amount of network parameters and most accurate prediction.

The resulting neural network prediction performance is shown in figure 5.4.

The trained neural network performs very well and will be utilized as a model for in a demonstration of the MIMO Nonlinear Predictive Controller developed in section 5.1.

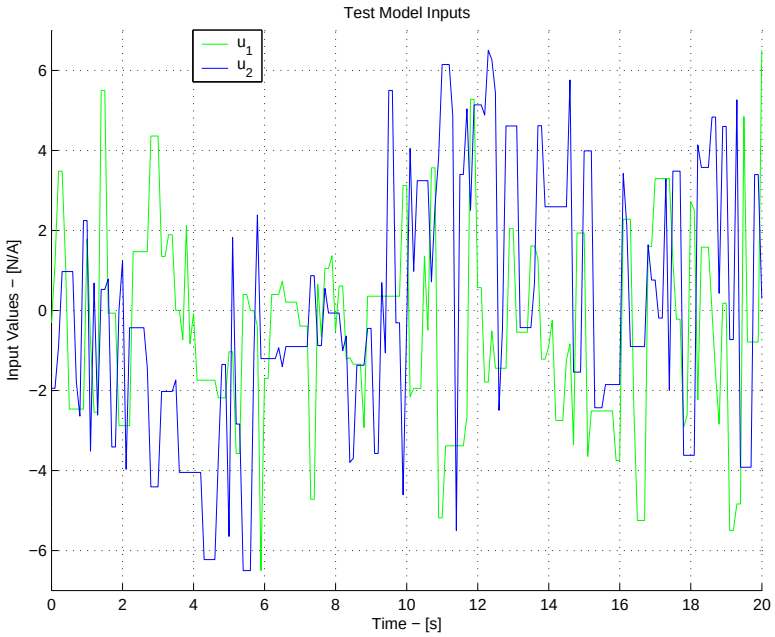


Figure 5.3: MIMO NPC Test Model Input Signals

Prediction Horizon	10
Hidden Neurons	7
Inputs	$u_{1,k}, u_{1,k-1}, u_{2,k}, u_{2,k-1}$ $y_{1,k}, y_{1,k-1}, y_{2,k}, y_{2,k-1}$
Output	$y_{1,k+1}, y_{2,k+1}$

Table 5.1: Test System Neural Network Model Training Parameters

5.2.3 MIMO Nonlinear Predictive Controller Demonstration

The test model and the MIMO NPC are set up as shown in figure 5.5.

The MIMO NPC demonstration is run in Matlab using the Matlab MEX function NPC (See appendix A.8.5) with the parameters shown in table 5.2. The resulting simulation output can be seen in figure 5.6.

Figure 5.6 clearly shows excellent control of the nonlinear MIMO test model. Notice the change in control output before the reference actually changes. This is characteristic for a predictive controller.

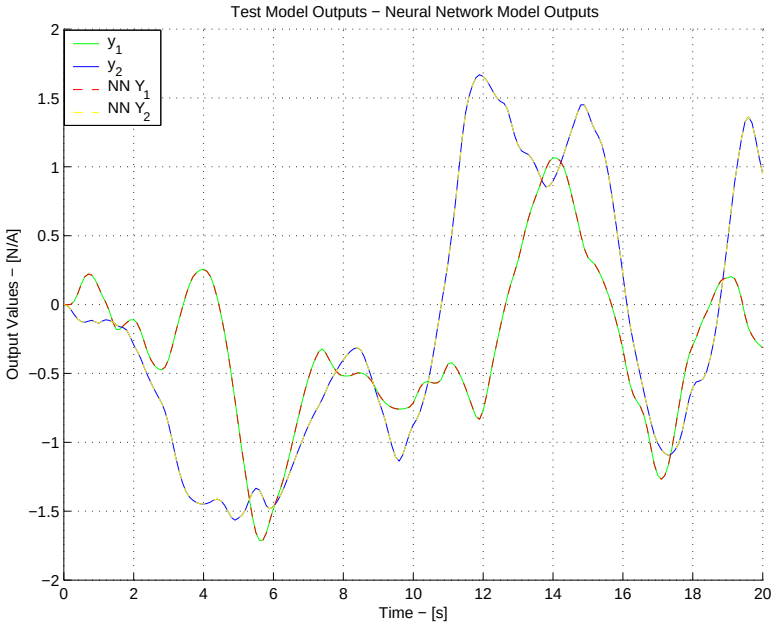


Figure 5.4: MIMO NPC Test Model and Neural Network Output Signals

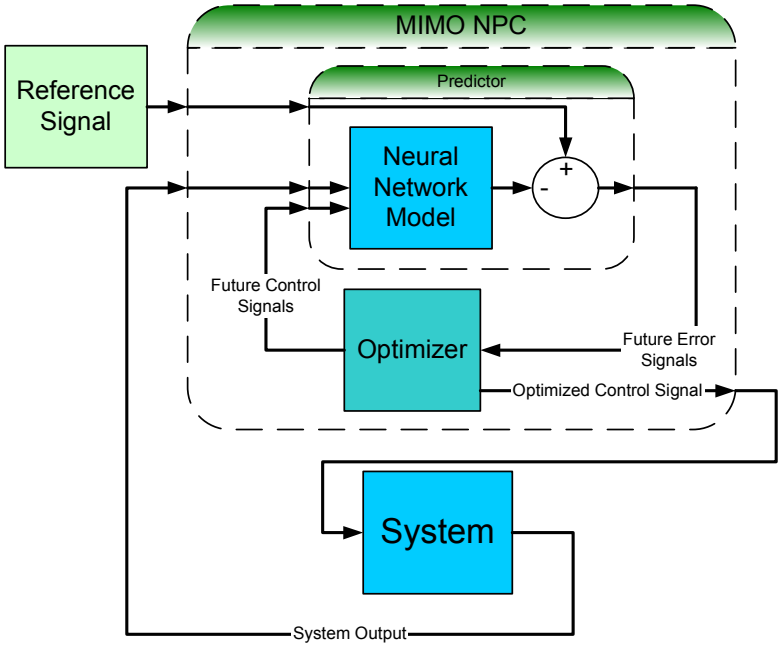


Figure 5.5: MIMO NPC Control Demonstration Set Up

UOrder	2
YOrder	2
HorizonStart	1
HorizonEnd	5
ControlHorizon	2
MaxIter	20
Rho	0.001
T (Sampling P.)	0.1

Table 5.2: MIMO NPC Control Demo Parameters

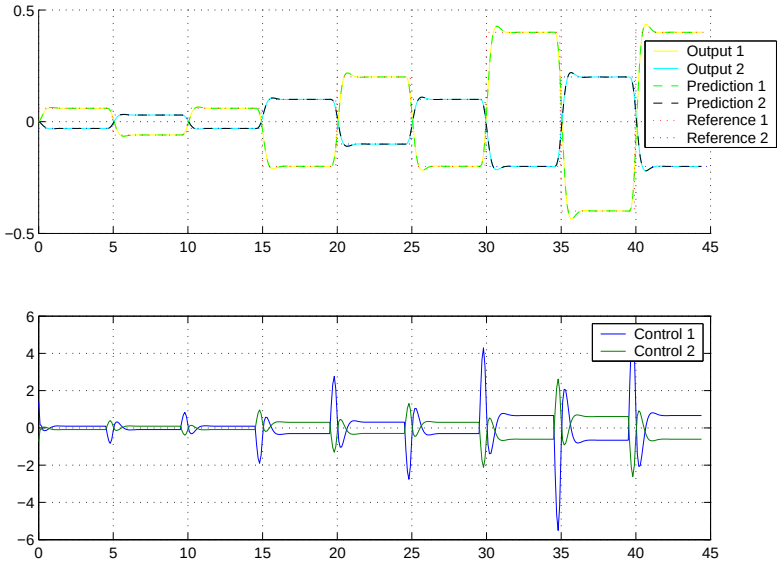


Figure 5.6: MIMO NPC Control Demonstration on the Test Model

5.2.4 Comparison With an H_2 Controller

A discrete H_2 controller will be constructed in order to evaluate the performance of the MIMO NPC compared to an advanced linear control method. The control signals and test system output for both controllers will be analyzed.

Linearized State Space Form

The nonlinear test system equations in equation 5.74 has to be linearized and put into state space form in order to design an H_2 controller.

The state space form of the test system equations in 5.74 is

$$\begin{array}{l} x_1 = y_1 \\ x_2 = \dot{y}_1 \\ x_3 = y_2 \\ x_4 = \dot{y}_2 \end{array} \Rightarrow \dot{x} = f(x,u) = \begin{bmatrix} x_2 \\ -x_2 + x_3 - x_1 - {x_1}^3 + u_1 \\ x_4 \\ -x_4 + x_1 - x_3 - {x_3}^3 + u_2 \end{bmatrix} \tag{5.75}$$

Where

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{5.76}$$

The linearized state space matrices is easily derived from equation 5.75 and yields the following state space matrices.

$$
\begin{aligned}
A = \frac{\partial \mathrm{f(x,u)}}{\partial \mathrm{x}} &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1-3X_1(0)^2 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & -1-3X_3(0)^2 & -1 \end{bmatrix} \\
B = \frac{\partial \mathrm{f(x,u)}}{\partial \mathrm{u}} &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \\
C &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\
D &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}
\end{aligned}
\tag{5.77}
$$

H_2 Controller Setup

The H_2 controller setup utilizes the well known general form shown in figure 5.7.

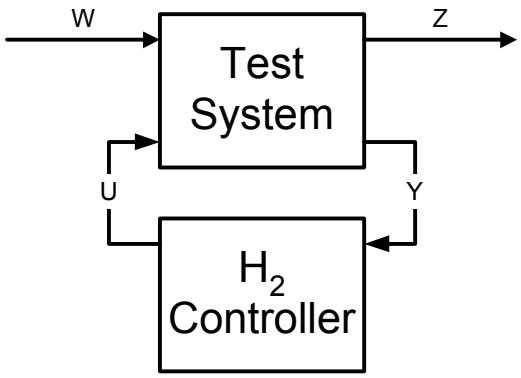


Figure 5.7: H_2 Controller Set Up

Where

- W is a vector input containing all disturbances.
- Z is a performance vector containing all the signals that should be small such as the tracking error.

- U is the system input.
- Y is the system output.

The following signals will be assigned to the vectors W, Z, U and Y.

$$\begin{aligned} W &= \begin{bmatrix} r_1 & r_2 \end{bmatrix}^T \\ Z &= \begin{bmatrix} fe_1 & fe_2 & fu_1 & fu_2 \end{bmatrix}^T \\ U &= \begin{bmatrix} u_1 & u_2 \end{bmatrix}^T \\ Y &= \begin{bmatrix} y_1 & y_2 \end{bmatrix}^T \end{aligned} \tag{5.78}$$

Where

- r_1, r_2 are the reference signals for the two test system outputs y_1, y_2.
- u_1, u_2 are the test system input signals.
- y_1, y_2 are the test system output signals.
- fe_1, fe_2 are the filtered error signals ($e_1 = r_1 - y_1, e_2 = r_2 - y_2$).
- fu_1, fu_2 are the filtered test system input signals (u_1, u_2).

$$\begin{aligned} fe_1 &= H_{ef1}(z^{-1})e_1 \\ fe_2 &= H_{ef2}(z^{-1})e_2 \\ u_1 &= H_{uf1}(z^{-1})u_1 \\ u_2 &= H_{uf2}(z^{-1})u_2 \end{aligned} \tag{5.79}$$

$$\begin{aligned} H_{ef1}(z) &= \frac{T}{z-1} \\ H_{ef2}(z) &= \frac{T}{z-1} \end{aligned} \tag{5.80}$$

$$\begin{aligned} H_{uf1}(z) &= \frac{\rho(z-1)}{z - e^{-\frac{T}{T_{ufs1}}}} \\ H_{uf2}(z) &= \frac{\rho(z-1)}{z - e^{-\frac{T}{T_{ufs2}}}} \end{aligned} \tag{5.81}$$

The error filters (equation 5.80) have been chosen to be discrete integrators in order to keep the stationary error as small as possible.

ρ	0.016
T_{uf1}	0.1
T_{uf2}	0.1

Table 5.3: H_2 Controller Parameters

The control signal filters (equation 5.81) have been chosen to be limited differentiating filters in order to keep the high frequency contents in the control signals low as well as not weighting the control signal in the stationary state. ρ is a weight on the control filter and is utilized as a weighting between the integrated error and the differentiated control signal output.

H_2 and MIMO NPC Controller Comparison

The NPC controller has been tuned to optimally with the MIMO NPC control parameters listed in table 5.2.

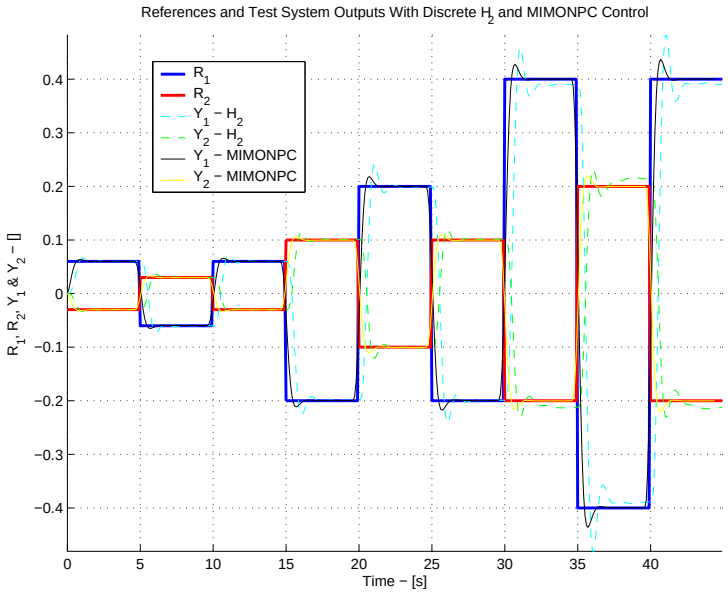
The H_2 controller has been tuned optimally while keeping the magnitude of the control signal in the same size as the MIMO NPC in order to make a comparison fair.

The control parameters for the H_2 controller are listed in table 5.3.

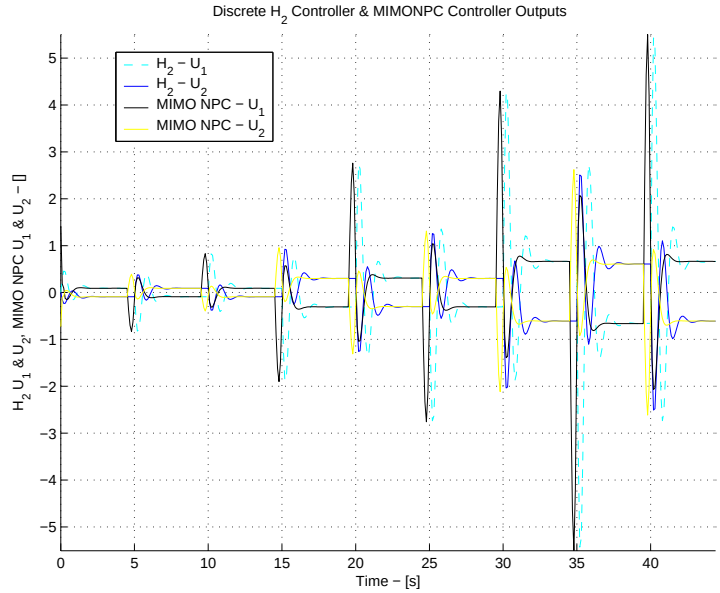
The simulation results for the H_2 controller can be found in figure 5.8(a) and 5.8(b). Figure 5.8(a) shows the test system outputs for the MIMO NPC and the H_2 controller in the same figure for easier comparison and figure 5.8(b) also shows the control signals for MIMO NPC and the H_2 controller in the same figure.

Figure 5.8(a) clearly shows smaller overshoots and stationary errors given the same amount of freedom with respect to the magnitude of the control signal. It handles the nonlinearities much better than the linear controller which is of course not surprising.

The predictive nature of the MIMO NPC is clearly seen near each step in the reference in figure 5.8(a). The test system outputs (when controlled by the MIMO NPC) "fit" the references much closer and is something the standard H_2 controller cannot do.



(a) MIMO NPC and H_2 Controlled Test System Outputs



(b) MIMO NPC and H_2 Control Signals

Figure 5.8: MIMO NPC and H_2 Controller Comparison

5.2.5 Conclusions

A neural network based predictive controller algorithm capable of handling non-linear MIMO systems has been developed. It is also referred to as the MIMO NPC.

A C++ implementation of the MIMO NPC algorithm has been programmed. Furthermore, the C++ implementation has been demonstrated on a test nonlinear MIMO system. The test showed perfect predictive control of a nonlinear MIMO system.

The MIMO NPC has been compared with a linear H_2 controller and the comparison showed an increased performance with smaller overshoots, smaller stationary errors and a tighter fit to the reference signals.

Chapter 6

Neural Network Based Controller Stability

Stability is always difficult when it comes to nonlinear controllers. Lots of assumptions has to be made and they are often very unrealistic. It is, however, still interesting and reassuring for those who would like to use the new advanced methods available to see that there are some indications of reliability.

In this chapter an extension to the nonlinear predictive controller strategy for the SISO case is presented.

This chapter is based on a paper by the author of this dissertation accepted at the IJCNN Conference in Washington 1999.

6.1 Introduction

Predictive control of non-linear systems has become increasingly interesting because good stability(see [3], [35], [47]) and robustness(see [29] and [2]) properties can be proven. These proofs are relatively general in the sense that only general properties of the non-linear system are required. The stability proofs are given both in continuous [28], [29] and [20] and discrete-time [3], [2],[35] and [47]. In particular the discrete-time stability guarantee is interesting in connection with real-time implementations.

In a number of papers the use of receding horizon control with neural network models or controllers has been investigated [20],[35] [47] and [36]. In particular, the paper [36] develop a fully implementable algorithm for a Generalized Predictive Controller GPC, with a single hidden layer feed forward neural network as a

model of the non-linear plant([36]). The minimization of the GPC cost function has to be performed numerically since the target systems are usually non-linear systems. In [36] the numerical optimization algorithm is developed in detail and convergence of the numerical algorithm is emphasized. However, the stability of the resulting controller cannot be guaranteed. The present paper addresses this problem, and gives a proof of asymptotic stability.

In [36] the system model is a non-linear single input single output model but for convenience it is assumed here that the system to be controlled is non-linear, time-invariant and described by a discrete time state equation of the form.

$$x_{k+1} = f(x_k, u_k) \ , \ \forall x_k \in X \ , \ \forall u_k \in U \tag{6.1}$$

Where $f \in \mathcal{C}$ (the set of continuous vector-functions on $X \times U$), $f(0,0) = 0$, X and U are compact sets on $\mathcal{R}^n$ and $\mathcal{R}^m$ respectively, and they include the origin.

If the function f is formed from a single hidden layer feed forward neural network with continuous activation functions the continuity properties of f are guaranteed.

The neural network based GPC cost function in [36] is

$$J(k, u(k)) = \sum_{i=N_1}^{N_2} [r(k+i) - \hat{y}(k+i)]^2 + \rho \sum_{i=0}^{N_u} \Delta u(k+i)^2 \tag{6.2}$$

It is seen to be a quadratic cost function of the system error (reference minus model output) summed over the time interval from N_1 to N_2 in the future, Also the cost function includes the future control signal changes ($\Delta u_k = u_k - u_{k-1}$) over the control horizon (N_u). Paper [36] gives a detailed account of the implementation of the optimizing controller. This cost function makes it possible to reduce the number of control signals(N_u) to compute independently of the prediction horizon. Moreover the prediction does not have to begin at sample k, but can be moved beyond the systems time delay. The cost function is not in the form assumed in the paper [47] and the stability of proof is thus not immediately obvious for the cost function in equation 6.2.

6.2 The Proof

The proof of stability is based on rewriting the cost function and system model in such a form that the general proof of stability in [47] can be utilized. This can be achieved by only minor changes in the assumptions.

6.2.1 The Cost Function

First a few changes to (6.2) are necessary. The outputs in (6.2) are replaced with the system states to match the setup in [47]. This is done for the sake of simplicity. This does not impair the generality of the scheme because the outputs are merely a mapping of the system states.
It will also be assumed that the equilibrium point of the system is located in the origin. This will not impair the generality of the scheme either since one can always make a state transformation to make the origin an equilibrium point of the new system. Let for instance the system

$$\bar{x}_{k+1} = \bar{f}(\bar{x}_k, \bar{u}_k) \tag{6.3}$$

have an equilibrium point ($\bar{f}(\tilde{x}, \tilde{u}) = \tilde{x}$) in $\bar{x} = \tilde{x}, \bar{u} = \tilde{u}$, then a new system that satisfies the assumptions can be found by defining a new state and a new control signal.

$$\begin{aligned} x_k &= \bar{x_k} - \tilde{x} \\ u_k &= \bar{u_k} - \tilde{u} \\ x_{k+1} &= f(x_k, u_k) = \bar{f}(x_k + \tilde{x}, u_k + \tilde{u}) - \tilde{x} \end{aligned} \tag{6.4}$$

The cost function (6.2) is supplemented with a final state penalty. This is necessary to ensure stability. Also the cost function is generalized by replacing the quadratic terms with positive functions, hereby obtaining the cost function:

$$J(x_k, N_1, N_2, N_u) = \sum_{i=N_1}^{N_2-1} h_x(x_{k+i}) + \sum_{i=0}^{N_u-1} h_u(u_{k+i}) + a\|x_{k+N_2}\|_P^2 \tag{6.5}$$

Where

1. $h_x, h_u \in \mathcal{C}$,
2. $h_x(0) = h_u(0) = 0$
3. $h_x(x) \geq 0,\ \forall x \in X_0 - \{0\}$
4. $h_u(u) \geq 0\ \forall u \in U - \{0\}$,
5. $a \in \mathcal{R}_+, \|x\|_P^2 = x^T P x, P > 0$

As in [36], the control signal u_{k+i} follows some predetermined sequence for $N_u \leq i \leq N_2 - d$, where d is the system time delay.

6.2.2 Assumptions

The assumptions needed to guarantee stability are basically the same as in [47] , but there are a few changes to accommodate the extra h-function $h_u(u)$.

Assumption 1:

The linearized system
$(A, B) = \left(\frac{\partial f}{\partial x}\,|_{x=0,u=0}, \frac{\partial f}{\partial u}\,|_{x=0,u=0}\right)$ is stabilizable.

Assumption 2:

$$\begin{array}{l} r_x(\|x\|^2) \leq h_x(x) \leq s_x(\|x\|^2) \\ r_u(\|u\|^2) \leq h_u(u) \leq s_u(\|u\|^2)\ ,\ \forall x \in X\ ,\ \forall u \in U \end{array} \tag{6.6}$$

where $r_x, s_x, r_u, s_u \in \mathcal{C}$ and are strictly increasing and
$r_x(0) = s_x(0) = 0,\ r_u(0) = s_u(0) = 0$

Assumption 3:

There exists a compact set $X_0 \subseteq X$, which includes the origin, with the property that there exists a control horizon $M \geq 1$ such that there exists a sequence of admissible control vectors.

$$\{u_k, \ldots, u_{k+M-1}, u_{k+M}, \ldots, u_{k+N_2-d}\} \tag{6.7}$$

that yield an admissible state trajectory

$$\{x_k, \ldots, x_{N_2}\} \tag{6.8}$$

ending in the origin.

Here

$$\{u_{k+M}, \ldots, u_{k+N_2-d}\} \tag{6.9}$$

is a predetermined sequence. Usually

$$u_{k+M+i} = u_{k+M+i-1}\ ,\ 0 \leq i \leq N_2 - d \tag{6.10}$$

that u_k follows for $k \geq N_u$.

Assumption 4:

The optimal control signal $u_{k+i},\ 0 \leq i \leq N_2$ is continuous with respect to x_{k+1}.

The trajectory in assumption 3, that u_k follows for $k \geq N_u$, is determined by the user of this algorithm. This information is needed by the minimization algorithm used, since the value of the states are used in the cost function for $k \geq N_u$. This trajectory usually reflects the stationary behavior of the system to be controlled.

6.2.3 Rewriting the Cost Function

Now all the assumptions necessary have been made to make use of the proof in [47] , and here the cost function (6.5) will be written in same form as in (6.13).

For given N_1, N_2 and N_u define

$$h(x_{k+1}, u_{k+1}) = I_x(i, N_1, N_2)h_x(x_{k+1}) + I_u(i, N_u)h_u(u_{k+1}) \tag{6.11}$$

where

$$\begin{aligned} I_x(i, N_1, N_2) &= \begin{cases} 1 & N_1 \leq i \leq N_2 - 1 \\ 0 & Otherwise \end{cases} \\ I_u(i, N_u) &= \begin{cases} 1 & 0 \leq i \leq N_u - 1 \\ 0 & Otherwise \end{cases} \end{aligned} \tag{6.12}$$

This will make the cost function in (6.5) equal to

$$J(x_k, N) = \sum_{i=0}^{N-1} h(x_{k+i}, u_{k+i}) + a\|x_{k+N}\|_P^2 \tag{6.13}$$

for $N = N_2$. By the new assumptions.

$$\begin{aligned} h(0,0) &= I_x(i, N_1, N_2)h_x(0) + I_u(i, N_u)h_u(0) \\ &= I_x(i, N_1, N_2)0 + I_u(i, N_u)0 \\ &= 0 \end{aligned} \tag{6.14}$$

$h(x, u)$ is continuous in x and u since $h_x(x)$ and $h_u(u)$ are.

For all i, (6.6) gives

$$\begin{aligned} h(x,u) \leq & I_x(i,N_1,N_2)s_x(\|x\|^2) \\ & + I_u(i,N_u)s_u(\|u\|^2) \\ \leq & s_x(\|x\|^2) + s_u(\|u\|^2) \\ \leq & s_x(\|(x,u)\|^2) + s_u(\|(x,u)\|^2) \\ = & s(\|(x,u)\|^2) \end{aligned} \tag{6.15}$$

$s(\|(x,u)\|^2)$ is continuous since $s_x(\|x\|^2)$ and $s_u(\|u\|^2)$ are, and $s(0) = s_x(0) + s_u(0) = 0$.

Noting that the cost $h(x,u) > 0, \ \forall x,u \neq 0$, the assumptions in [47] are fulfilled and the predictive control strategy minimizing the cost function in (6.5) are guaranteed stable.

6.2.4 A Neural Model

The function f in (6.1) could for instance be a neural network since the neural network output function is continuous.

$$x_{k+1} = f(x_k, u_k) = NN(x_k, u_k) \tag{6.16}$$

Where $NN(x_k, u_k)$ is a neural network output function trained to describe the system being considered. So if the system can be described by a neural network, this control is a stable and very useful strategy for nonlinear systems with significant time delays.

If the cost function (6.5) is utilized then the computational burden can be drastically reduced since the control signal usually becomes constant after a short while, and therefore does not need to be calculated explicitly by the minimization algorithm.

6.3 Example

In figure 6.1 and 6.2 is a graph of a simulation of a GPC controller in action on pneumatic servo system model. Figure 6.1 shows the response when utilizing the original cost function as in [36].

Figure 6.2 shows the response utilizing the cost function (6.5).

Both controllers are tuned as accurately as possible. This to see if there is any significant loss in performance or change in behavior by adding the final term cost. In both simulations, the following parameters were utilized.

$$N_1 = 1,\ N_2 = 10,\ N_u = 2,\ \rho = 0.05$$

As is seen in the figures, the performance is almost fully preserved for the controller with the new cost function. This means that the addition of the stabilizing term in the cost function does not significantly degrade the performance of the system.

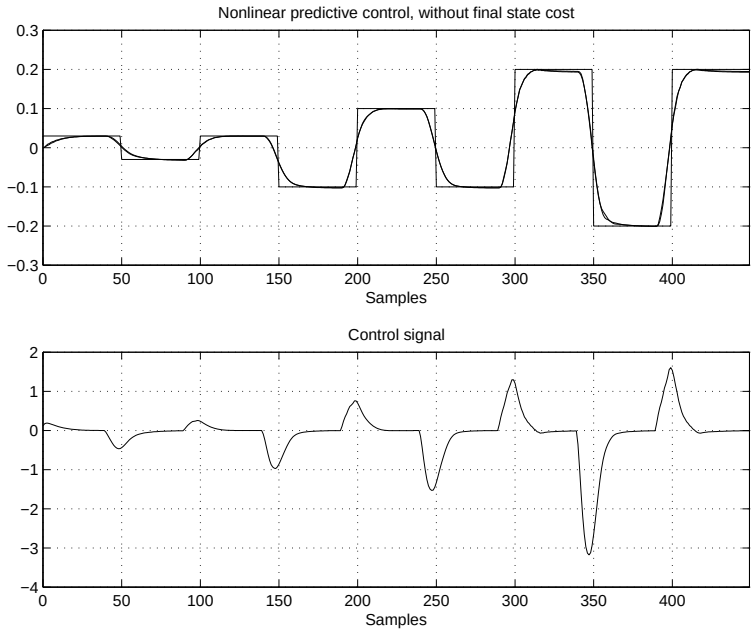


Figure 6.1: A simulation using the cost function without a final state cost added.

6.4 Conclusion

It has been found that the cost function (6.5) leads to a stabilizing control signal for a system that can be accurately modelled by a neural network. The system performance was only slightly reduced when utilizing the cost function that guarantees stability.

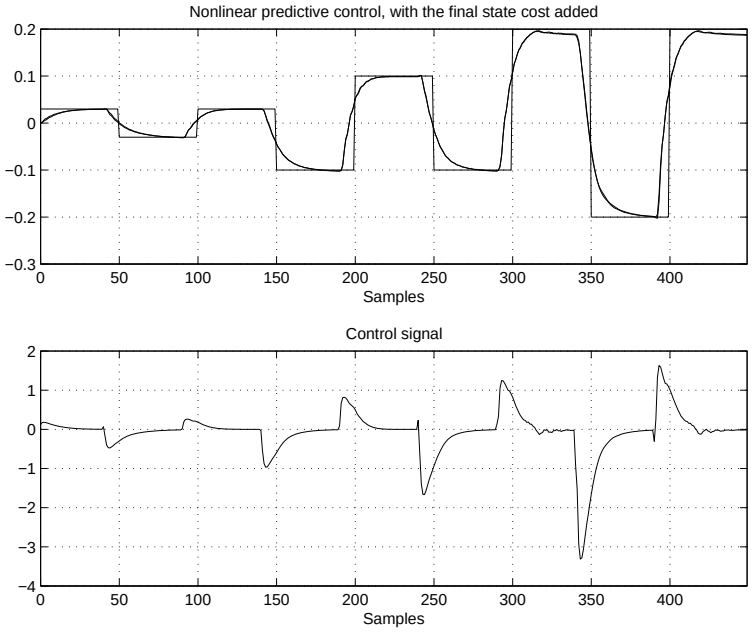


Figure 6.2: A simulation using the cost function with the final state cost added.

The next step will be to examine the consequences of utilizing a neural network model that does not model the system accurately.

Chapter 7

Overall Conclusions

7.1 Conclusion

7.1.1 Neural Network Virtual Sensors

Neural networks has been tested as virtual sensors based on in-cylinder pressure signals from a diesel engine in chapter 2. Several different kinds of virtual sensors were tried. Those were.

1. Exhaust O_2 concentration.
2. In-Cylinder Air Fuel Ratio.
3. Peak Pressure and Peak Pressure Location.

Exhaust O_2 Concentration

The neural network exhaust O_2 concentration estimation was quite good when trained and tested on the no EGR data sets (EGR was turned off during the sampling of these data) and the exhaust temperature was utilized as an extra input to the neural network (see figure 2.4 and 2.5). The largest test set error was in that case about -0.4 %[O_2]. The largest O_2 concentration estimation error increases greatly to about -2.8 %[O_2] for the test set when the exhaust temperature is not utilized.

The neural network virtual sensors were much more difficult to train when EGR was introduced and it seems like the exhaust temperature does not help improve the result in this case (see figure 2.6 and 2.7). The largest O_2 concentration estimation error for the test was about 2.8 %[O_2] numerically when utilizing the

exhaust temperature as a neural network input and about 4.5 %[O_2] when the exhaust temperature was not utilized as a neural network input.

The above mentioned errors are peak errors for the test data sets and the overall performance of the neural network exhaust O_2 concentration sensors is ok although not yet good enough to replace a Lambda sensor. The problems are believed to be due to the large sensor noise coming from thermal shock which makes some of the pressure features extracted from the pressure data very inaccurate and thus poorly correlated with the target output.

In-Cylinder Air Fuel Ratio

The neural network in-cylinder air fuel ratio virtual sensor also had some problems with a maximum test set error of about $\pm 38\%$. Those maximum errors are only in a few peaks and the overall picture is not that bad, although still not good enough to replace a lambda sensor. The test set relative errors can be seen in figure 2.10. The error is for the most part smaller than $\pm 10\%$.

The virtual in-cylinder air fuel ratio sensor neural network was in this case also trained utilizing the same in-cylinder pressure features as for the exhaust O_2 concentration. And those features are due to the pressure sensors noise level not good enough to provide the necessary correlation level for good virtual neural network sensor training.

Peak Pressure and Peak Pressure Location

The peak pressure neural network virtual sensor performed fairly well with an error of about ± 2 bar for the most part. A single spike in the error reaches 3 bar. The pressure was sampled at a resolution of 6 crank shaft angle degrees which still makes the remaining error somewhat large compared to the relative small changes (about 8 bars in the area near the pressure peak) in the pressure samples at a resolution of 6 degrees. The error could however not be made smaller and it is believe that this is because of sensor noise.

The peak pressure location neural network virtual sensor was not successful with the available data for this work. The maximum test set error reached 20 degrees and is unacceptable. The data was full of odd outliers which the author has not been able to explain by anything else than sensor errors. It is however believed that the results can be improved by either acquiring new data sets or by cleaning up the existing ones, but the sensors noise is still a problem here too and should

be made smaller.

Overall Virtual Sensor Conclusions

The virtual sensor experiments looks promising, but sensor noise from current pressure sensors available for mass production appears to be a problem. To much data is lost in noise or disfigured in a way that makes it extremely complicated to find a suitable set of input signals and network structure.

Another possibility is that the in-cylinder pressure is physically just too complicated for this kind of neural network setup. Advanced signal processing of the in-cylinder pressure signal and a detailed mathematical model of the in-cylinder pressures behavior might be necessary in order to improve the results.

The author believes that better in-cylinder pressure sensors and/or a physical/mathematical study of how the in-cylinder pressure behaves the way to improve the neural network virtual sensors.

Software

The following tools for training the pressure feature based neural network virtual sensors has been developed.

1. A MATLAB graphical user interface for neural network training. See appendix A.2.
2. A pressure feature extraction program. See appendix A.3.
3. An INDISET data file conversion program. See appendix A.4.

7.1.2 Dynamic Neural Network Training

Predictive Training Algorithm

A predictive training algorithm was developed which takes the predicted errors up until a specified horizon into the cost function when training the neural networks. This proved to be very valuable when training dynamic neural network models since it reduces the chances of ending up with an unstable model and helps increase the accuracy of the neural network prediction. A demonstration of the predictive training algorithm is also given in which the predictive training algorithm

clearly outperforms the standard one step ahead training algorithm commonly utilized.

It is especially valuable for training neural networks utilized in a predictive controller since it usually has a short prediction horizon. The prediction horizon given to the predictive training algorithm is usually short too because of the large memory consumption and training time necessary when training with long prediction horizons. A very small neural network prediction error within the prediction horizon is however almost always the result of training neural network with the predictive training algorithm and this is important for the predictive controller.

Software

The following neural network training software has been developed.

1. A matrix C++ library for easy implementation of matrix operations in the software developed for this work.
2. A Levenberg-Marquardt optimizer C++ class.
3. A neural network C++ library containing a single hidden layer neural network class, A regular one step ahead neural network training class and a predictive neural network training class. The neural network library makes use of the matrix library and the Levenberg-Marquardt optimizer class.

7.1.3 Neural Network Engine Modelling

A new β_2 (see equation 4.12) function has been suggested to help the AMVEM model handle intake manifold pressures larger than ambient pressure and the infinitely large gradient of β_2 at a pressure ratio of one. A limited gradient improves the simulation speed of AMVEM models since the integration routines can more easily solve the problem.

A large error in the AMVEM modelling of the intake manifold temperature has been discovered during the work with dynamic neural network models. A plot containing the measured signals from the ECG test engine and the AMVEM simulated signals is presented.

A fallout removal algorithm which can remove most of the fallouts in the measured engine signals has been developed and implemented in MATLAB.

The throttle air mass flow neural network model found in this work was more accurate than the AMVEM version for the test sets similar in structure to the training set, but was equal to or slightly worse in some places on test sets that were not similar in structure to the training set.

The normalized air fuel ratio (lambda) neural network model turned out to be quite difficult to model. There were some peculiar spikes in the lambda signal which the neural network could not be made to model. It was argued that the spikes perhaps was a sensor fault and this might be the reason why the lambda neural network models accuracy was quite bad for some test sets. The lambda model was not successful, but it is believed that it can be made to work if the noise level of the lambda signal can be made smaller.

The intake manifold pressure model was an overall better model than the AMVEM model, but still showed the same kind of reduction in performance on test sets of a different type than the training set. The pressure model is one of the best models achieved in this work which is most like due to the relatively clean pressure signal. This shows the importance of having good sensors when modelling with neural networks.

The intake manifold temperature model was not quite as successful as the intake manifold pressure model. The accuracy on test set with the same structure as the training set is however consistently much better than the AMVEM models temperature output. The missing physics in the AMVEM that causes the large temperature error has been identified by the neural network model. The performance is however reduced on test sets of another type than the training set. The measured temperature signal is however in this work not completely reliable since it is an experimental type of fast temperature sensor. The performance of the neural network temperature model will of course also be affected by this fact.

The experimental fast temperature sensor utilized in this work is not reliable and the accuracy of the sensor is not yet known. The performance of the neural network models trained in this work which utilize the temperature signal (the lambda model, the intake manifold pressure model, the intake manifold temperature model) must thus be viewed in this light. The author of this work believes that the temperature sensor is too noisy at this time and that the high noise level has seriously reduced the performance of the neural network models.

7.1.4 Neural Predictive Controller

A neural network based predictive controller algorithm capable of handling nonlinear MIMO systems has been developed. It is also referred to as the MIMO NPC.

A C++ implementation of the MIMO NPC algorithm has been programmed. Furthermore, the C++ implementation has been demonstrated on a test nonlinear MIMO system. The test showed perfect predictive control of a nonlinear MIMO system.

The MIMO NPC has been compared with a linear H_2 controller and the comparison showed an increased performance with smaller overshoots, smaller stationary errors and a tighter fit to the reference signals.

7.1.5 Stability of Predictive Control Strategy

The stability for the type of predictive control cost function utilized in the predictive controller algorithm developed in this work is proven if a final state cost is added to the cost function.

7.2 Suggestions for the Future

7.2.1 Sensors

The pressure sensors utilized for the virtual sensor training (chapter 2) are not reliable enough. The thermal shock is probably affecting the pressure features correlation with the target outputs too much. It makes the neural network training very hard since the information is then very hard to extract accurately from the pressure traces. It is suggested that these sensors are improved by creating filters capable of inverting the thermal shock effect.

The new temperature sensor utilized in the data gathering process for the neural network engine subsystem modelling (chapter 4) needs to be made less noisy. The noise level is unacceptably high and makes neural network training very difficult. Furthermore, the accuracy of the new temperature sensor must be found since it is otherwise impossible to know wether or not a correct neural network model or other types of models of the intake manifold temperature has been found.

The lambda sensor must be investigated to make sure that the strange peaks in the lambda sensor signal (see figure 4.18) really exist. For instance by measuring the

lambda value with three lambda sensors and then compare the outputs.

7.2.2 Data Sets

The neural network modelling training runs into the same problem all the time. The training data set is not sufficiently exciting to make the neural network model generalizing enough. All of the models trained in the mean value engine modelling chapter displayed too great a performance reduction when tested on a test data set of another type.

There were mainly two different types of data sets. The pyramidical type (see figure 4.11(a) and 4.12(a)) and the varied step type (see figure 4.11(b) and 4.12(b)). It is suggested to generate new types of data sets for improved neural network modelling in future works. Perhaps a combination of the two types of data sets.

7.2.3 Data Filtering

Many of the signals sampled from the engine is affected by pumping fluctuations and sensor noise. The neural network training is not working very well on this kind of data and filtering seems to be a requirement. Sensors will sometimes also give a completely wrong readout (a fallout) which also disturbs neural network training.

Two filters has been developed during this work as an attempt to automatically remove fallouts and a variance dependent filter to smooth out pumping fluctuation effects from the training data without smoothing important high frequency contents like the manifold filling spike.

The fallout removal filter worked fairly well and was applied on all data sets for this work as described in section 4.7.3. It does however still need development since it will have problems when there are fallouts in an area where the "support" signal is also active (a large step).

The variance dependent smoothing filter was not at this time working well enough to be utilized in this work, but the idea will be briefly described here as inspiration to future work.

Variance Dependent Filter

The signals throttle air mass flow, intake manifold temperature and the air fuel ratio contain a lot of noise and pumping fluctuations from the engine events. The simple way to remove this noise would be to low pass filter the signals, but this would for instance also filter away the manifold filling spike which contains important information.

The signal to be filtered will in the following be referred to as x.

The idea with the variance dependent filter is to calculate a running variance signal for x. The running variance signal is a signal generated by calculating the variance for the values of x in a window around each point in x. This will produce a signal that contains information about the local variance for each point in x. The variance window length is optional. The longer it is the smoother the running variance signal will be.

Each point in x is now replaced by the mean value of the samples in a window around the point. The mean value window length is determined by the magnitude of the running variance signal. The largest running variance signal value will correspond to a window length of 1 sample (which will return the value itself at that point). The smallest running variance signal value will correspond to a specified maximum window length.

This will not filter the large manifold filling spikes since they will have a large local variance, but the pumping fluctuations will have a smaller local variance and will thus be replace by mean values based on larger window lengths.

The source code for a suggestion to this filter can be found in the folder ToolBoxes/NeuralSysID on the source code appendix CD coming with this dissertation.

The graphs in figure 7.1 shows how the filter works on a throttle air mass flow signal sampled from the ECG Test Engine. The variance dependent filter is compared to a constant window length mean value filter where each point is simply replaced by the mean value of a constant number of samples around the point. The variance dependent filter follows the rising edge of the intake manifold filling spike better than the constant window length mean value filter. The variance dependent filter does however still needs improvement to make it match the filling spike better.

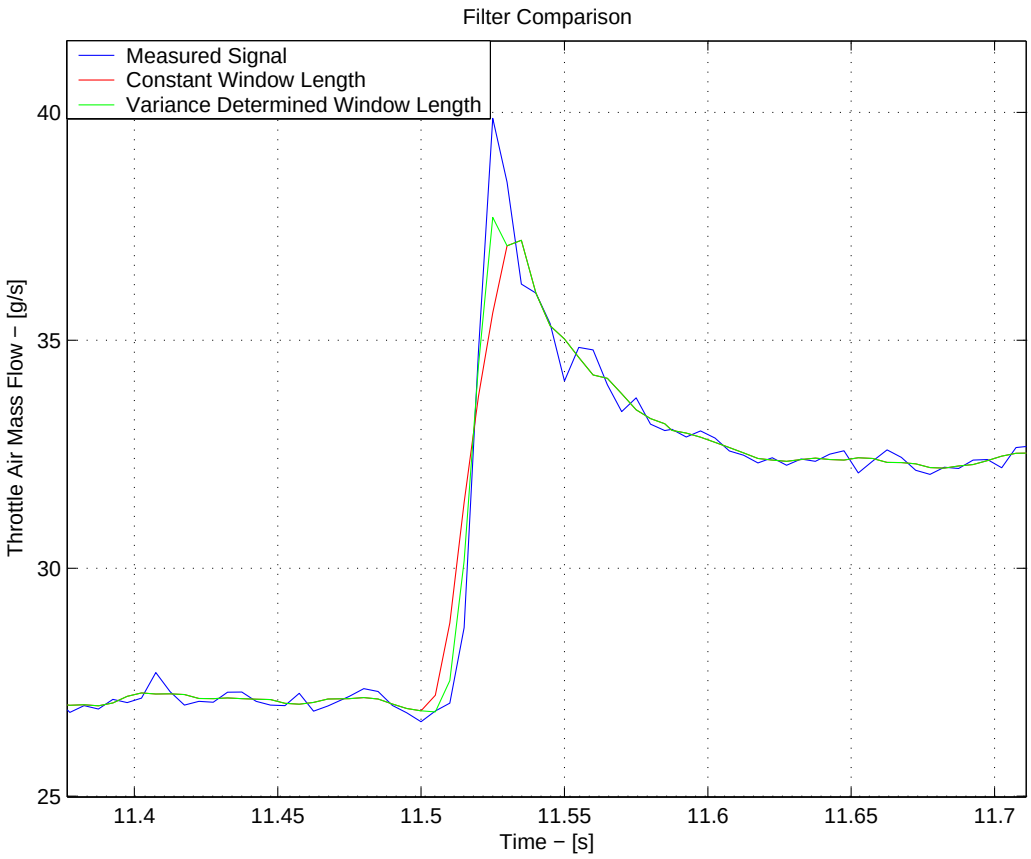


Figure 7.1: Variance Dependent Filter Demonstration

7.2.4 Predictive Training Algorithm

The predictive training algorithm works well, but it consumes a great amount of memory and is much slower than the regular one step ahead training algorithm since the error vector is ph (prediction horizon, see equation 3.9) times the number of system outputs times the number of input output sets long. The number of floating point operations quickly increasing with the size of ph.

It would be worth trying to make the predictive error vector (see equation 3.11 and 3.12, $E = Y - R$) shorter by making each element in the error vector the sum of squared prediction errors for one input output data set as shown in equation 7.1.

$$E = \begin{bmatrix} \sum_{i=1}^{ph} \left(r^{1}_{maxd+1+i-1} - \hat{y}^{1}_{maxd+1+i-1,i} \right)^2 \\ \vdots \\ \sum_{i=1}^{ph} \left(r^{n_{out}}_{maxd+1+i-1} - \hat{y}^{n_{out}}_{maxd+1+i-1,i} \right)^2 \\ \sum_{i=1}^{ph} \left(r^{1}_{maxd+2+i-1} - \hat{y}^{1}_{maxd+2+i-1,i} \right)^2 \\ \vdots \\ \sum_{i=1}^{ph} \left(r^{n_{out}}_{maxd+2+i-1} - \hat{y}^{n_{out}}_{maxd+2+i-1,i} \right)^2 \\ \sum_{i=1}^{ph} \left(r^{1}_{maxd+3+i-1} - \hat{y}^{1}_{maxd+3+i-1,i} \right)^2 \\ \vdots \\ \vdots \\ \sum_{i=1}^{ph} \left(r^{n_{out}}_{maxd+ndft+i-1} - \hat{y}^{n_{out}}_{maxd+ndft+i-1,i} \right)^2 \end{bmatrix} \tag{7.1}$$

This error vector is ph times shorter and will make the predictive training much faster. The derivatives will however be somewhat more complicated.

7.2.5 Neural Network Structure

It would be interesting to try out other neural network structures to see if they are better suited for engine modelling. The next step would be to implement multi layer neural networks. Those neural networks have more than one hidden layer of neurons. Figure 7.2 shows a two layer neural network.

This kind of neural network opens up for more structuring with respect to which hidden neurons that take inputs from where. The training can be made easier by assigning a certain structure to the network before the training. This could for instance be the division of the first hidden layer neurons into blocks each taking care of specific inputs or the division of the last layer of output neurons into blocks taking care of specific outputs. This would make a complex neural network with many neurons capable of having smaller networks inside the entire network capable of handling smaller less complex problems which will probably make the neural network training easier and more successful.

It is believed that structuring the neural networks neurons connection is especially useful when training multi output neural networks. The training of multi output neural networks is especially difficult with respect to the number iterations necessary and with respect to finding a good training set capable of making the neural network generalizing enough. The chance of the optimizing training process ending up in a bad minimum (with respect to estimation error and generalizing ability) is much higher for multi output neural networks because of the increased complexity.

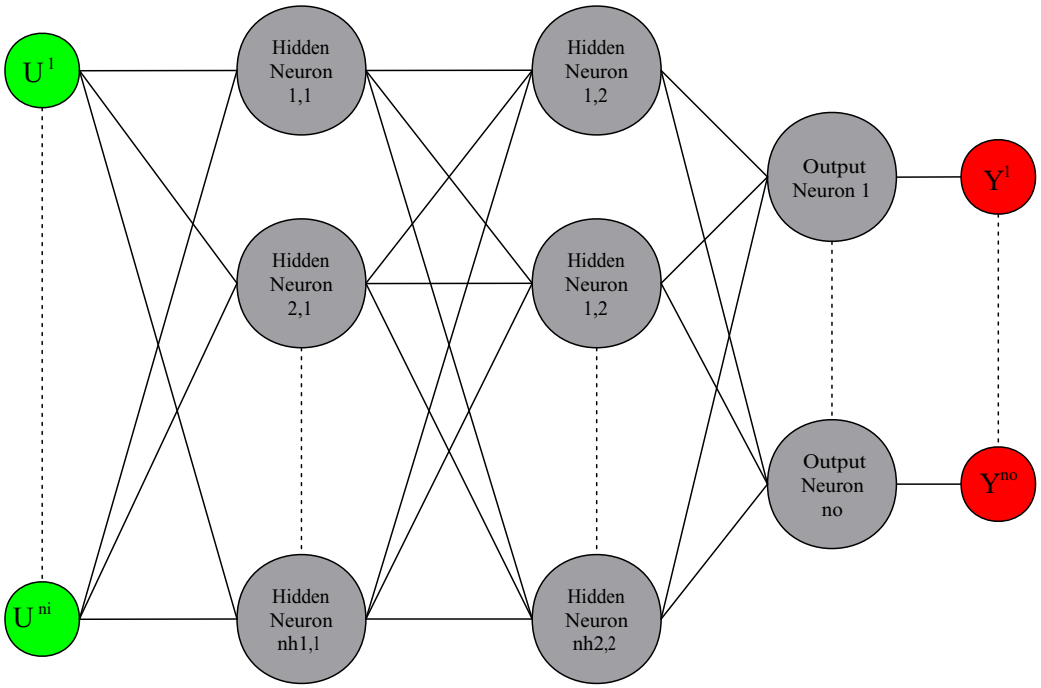


Figure 7.2: A Two Layer Neural Network

7.2.6 MIMO Nonlinear Predictive Control Algorithm

The MIMO nonlinear predictive control algorithm developed in chapter 5 does not utilize the final state cost which is necessary to guarantee stability under certain assumptions described in chapter 6. It should be examined how this can be implemented in the MIMO algorithm.

7.2.7 PredictiveController Class

The PredictiveController class as it is now is not capable of handling models with inputs that are not control inputs.

The intention with this work was to develop a neural network model of the entire engine that takes the throttle plate angle command and the fuel mass flow command as inputs and has the lambda value and the crank shaft speed as output. Such a model will also need to have the intake manifold pressure and temperature as extra information to help model the outputs accurately. The pressure in the intake manifold is correlated with the load and will be necessary to model the cranks shaft speed correctly.

The PredictiveController class are not currently coded to handle such a model. It is assumed that all the models inputs are control inputs. Furthermore, there are, as mentioned in the chapter Neural Engine Control, still some bugs in the code when a MIMO model is utilized. The derivative of the predictive error vector is not being calculated correctly.

Appendix A

Tools Used and Developed

It was necessary to develop some tools to process the data and to make it faster to experiment with different setups. These tools are described in this chapter. The source code for all the classes as well as some MATLAB mex functions not described here can be found on the source code appendix CD coming with this dissertation.

A.1 IAU Neural Network Toolbox

To perform the training for the neural network virtual sensor experiments in chapter 2, a neural network MATLAB toolbox developed by Peter Magnus Nørgaard at IAU (Department of Automation at DTU, Denmark) was utilized.

The toolbox is freeware and the author requires only the a reference to a technical report [32] about this toolbox is made in any published work in which the toolbox has been utilized. But he does require that a written consent is obtained from him if this is utilized for commercial products.

A.2 Neural network graphical user interface

A Graphical User Interface (GUI) for MATLAB was made In order to make it faster to try several different combinations of features as inputs to the neural network. The MATLAB source code for the GUI can be found in the ToolBoxes/NeuralPressure folder on the source code appendix CD coming with this dissertation.

Figure A.1 shows a screen shot of the GUI.

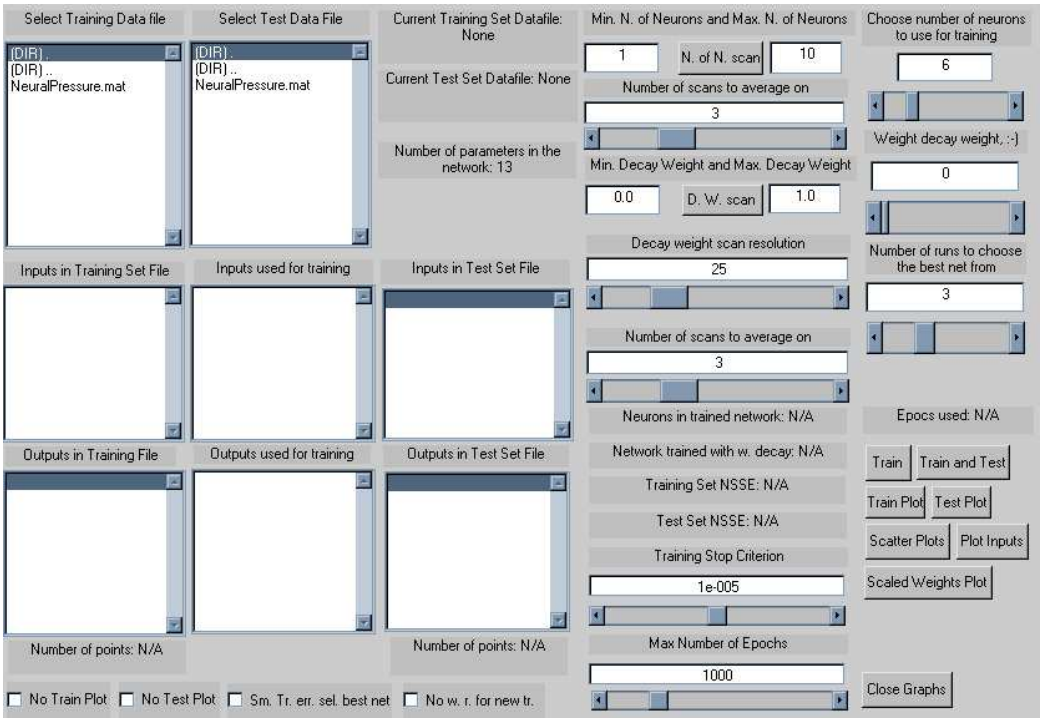


Figure A.1: Neural Network Toolbox Graphical User Interface

A.2.1 How to use it

Selecting data files

In this GUI you select two data files, a training data file and a test data file, whose format will be described later in this section.

These files are selected in the listboxes.

- Select Training Data File.
- Select Test Data File.

One can move around in the directory structure of the current drive by left clicking on the directories. Only .mat files are displayed in these list boxes.

When these files have been selected, the filenames will appear in the text boxes right next to the select test data file list box. The contents of the files will be displayed in the list boxes.

- Inputs in Training Set File.
- Outputs in Training Set File.
- Inputs in Test Set File.
- Outputs in Test Set File.

In these windows one can can see the available inputs to the network. By left clicking on one of the inputs in the list box for the inputs in the training set file, one can move it into the list box right next to it.

- Inputs used for training.

And the inputs selected will be used as inputs for the network during training and evaluation.

The same goes for the list box for the output in the training set file. Left clicking an output selects this output to be an estimation target for the network and moves it into the list box right next to it.

- Outputs used for training.

One can not choose an input for training that is not available in the test set file because the GUI would be able to evaluate the trained network on the test set.

The number of points available in the training set file and the test set file will be displayed in the text boxes below the outputs list boxes.

Setting network parameters

When the data files have been selected, then it is time to set the neural network parameters. These parameters are.

1. Number of neurons in the hidden layer.
2. Weight decay weight.

These parameters are set using either the edit box or the sliders labelled.

- Choose number of neurons to use for training.
- Weight decay cost function weight.

The number of neurons decide how much flexibility the neural network will have. The more neurons the more complex a mapping. But one should always have more data points than network parameters and each time a neuron is added more network parameters are introduced. The number of network parameters is displayed and updated as choices are made in the text box named.

- Number of parameters in the network.

The weight decay weight is the value of the elements in the diagonal in the D matrix in equation 1.3. This controls regularization as is described in paragraphs following equation 1.3.

Setting the training parameters

There are also some parameters that controls the training of the network. These are.

1. Training Stop Criterion.
2. Max. Number of Epochs.
3. Number of runs to choose the best net from.
4. Sm. Tr. err. sel. best net.

The training is stopped when either the max. number of epochs has been reached or when the estimations cost, which is 1.3 without the regularization term, is below the training stop criterion.

The number of runs to choose the net from simply determines how many times the network is trained. Each time the weights are initialized with random values to make the network weights fall into different local minima. The set of final weights that produces the best estimation is then chosen after all the runs has been completed.

The Sm. Tr. err. sel. best net.(smallest training error selects the best network) determines whether the best network after a number of training runs is chosen based on the test set estimation error or the training set estimation error.

Training the network

The actual training of the network can be started in two ways.

- Using the Train button.
- Using the Train and Test Button.

The difference is that when using Train, no evaluation on the test set is performed. For test set evaluation right after training, one can press the Train and Test button.

Normally two plots for both the training and the testing of the network is produced. But the following checkboxes control this behavior.

- No Train Plot.
- No Test Plot.

Checking these checkboxes turns off the plots.

The final training and test error can be seen in the text boxes named.

- Training Set NSSE.
- Test Set NSSE.

And the two text boxes.

- Neurons in trained network.
- Network trained with w. decay.

Displays the network parameters for the last trained network.

Finally. If one wants to continue training on a network that has been trained with regularization($D \neq 0$) then can check the checkbox.

- No w. r. for new tr.

And the weights are not initialized with random values as normally when a new training is started.

Network parameter utilities

There are also some utilities in the GUI to help find the network parameters. These are.

1. A tool for finding the best number of neurons.
2. A tool for finding the best values of the weight decay weight.

The first trains a network on the selected data several times. First using a minimum number of neurons and then one more, and so on until the maximum number of neurons has been reached.

The training and test errors are recorded and when a chosen number of neuron sweeps has been performed, a graph will appear displaying the, over the number of sweeps, averaged training and test errors as a function of the number of neurons. One can then based on this graph choose the best number of neurons.

The controls for finding the best number of neurons are named.

1. The edit box : Min. N. of Neurons.
2. The edit box : Max. N. of Neurons.
3. The button : N. of N. scan.
4. The edit box and slider : Number of scans to average on.

Number 1. and 2. controls the range of neurons to examine.
Number 3. Starts the scan.
Number 4. controls the number of sweeps to perform.

The second tool trains a network on the selected data several times. First using a minimum decay weight value and then the next, and so on until the maximum decay weight value has been reached.

The training and test errors are recorded and when a chosen number of decay weight sweeps has been performed, a graph will appear displaying the, over the number of sweeps, averaged training and test errors as a function of the decay weight. One can then based on this graph choose the best decay weight value.

The controls for finding the best value of the decay weight are named.

1. The edit box : Min. Decay Weight.

2. The edit box : Max. Decay Weight.

3. The button : D. W. scan.

4. The edit box and slider : Decay weight scan resolution.

Number 1. and 2. control the range of the decay weight.
Number 3. starts the scan.
Number 4. controls the number of decay weight values to examine in each sweep.

A.2.2 Train and test file format

The GUI expects the training and test set data files to be .mat files containing some variables with certain names and structure.

The training set file and the test set file is a .mat file containing four variables. These four variables must be named and constructed as specified in this section.

Neural Pressure Data File Matrix Format		
Variable Name	Type	Dimensions
InNames	Cell Array	Number of inputs x 1
OutNames	Cell Array	Number of outputs x 1
networkin	Matrix	Number of points x Number of inputs
networkout	Matrix	Number of points x Number of outputs

Table A.1: Neural Pressure Data File Matrix Format.

The variables should be filled with contents in the following format.

$$
\begin{aligned}
InName &= \{'InputName1','InputName2',.....,'InputNameN'\} \\
OutName &= \{'OutputName1','OutputName2',.....,'OutputNameM'\} \\
networkin &= \begin{bmatrix} \text{Input 1-Point 1} & \cdots & \text{Input N-Point 1} \\ \vdots & & \vdots \\ \text{Input 1-Point NP} & \cdots & \text{Input N-Point NP} \end{bmatrix} \\
networkout &= \begin{bmatrix} \text{Output 1-Point 1} & \cdots & \text{Output N-Point 1} \\ \vdots & & \vdots \\ \text{Output 1-Point NP} & \cdots & \text{Output N-Point NP} \end{bmatrix}
\end{aligned}
\tag{A.1}
$$

A.3 The extraction program

A MATALB program called Extraction has been made to produces the data files with the pressure features listed in table 2.2 in the format explained in section A.2.2. It asks for a directory containing the pressure data files and the stationary data file (in text format) and for an output directory and filename. The text file containing the stationary data must have two lines in the beginning with the signal names and the units in columns. The data follows in the remaining lines in columns below their name and unit. The names of the stationary value listed in the first line of the text file will become the variable names in the MATLAB workspace and specific names are required for the extraction program to work correctly. Engine speed has to be named N for instance. The rest of the names can be seen in the file extract.m. The MATLAB files for this program can be found on the source code appendix CD in the folder ToolBoxes/Extraction.

A screen shot of the program interface is shown in figure A.2.

The source directory should contain the following two directories using exactly these names.

1. Pressure Traces.

2. Stationary.

The Pressure Traces directory contains all the .mat files that the program PCON (See the next section A.4) has generated. These files contain all the pressure traces converted into .mat file format.

The extraction program is current only uses one of the pressure traces and that is hard coded into the extraction.m file.

The Stationary Data directory must contain a .puma tab delimited text file that has all the stationary data in it. One particularly important information it must contain is the filename of the the corresponding pressure trace file and the file number.

This is currently a little messy since the PUMA system decided to save the filename twice right after each other under the same variable name INM_FNR. The file number is under the variable name INM_FNR.

But the way the pressure traces files are chosen from the stationary data file can be studied in the extraction.m file.

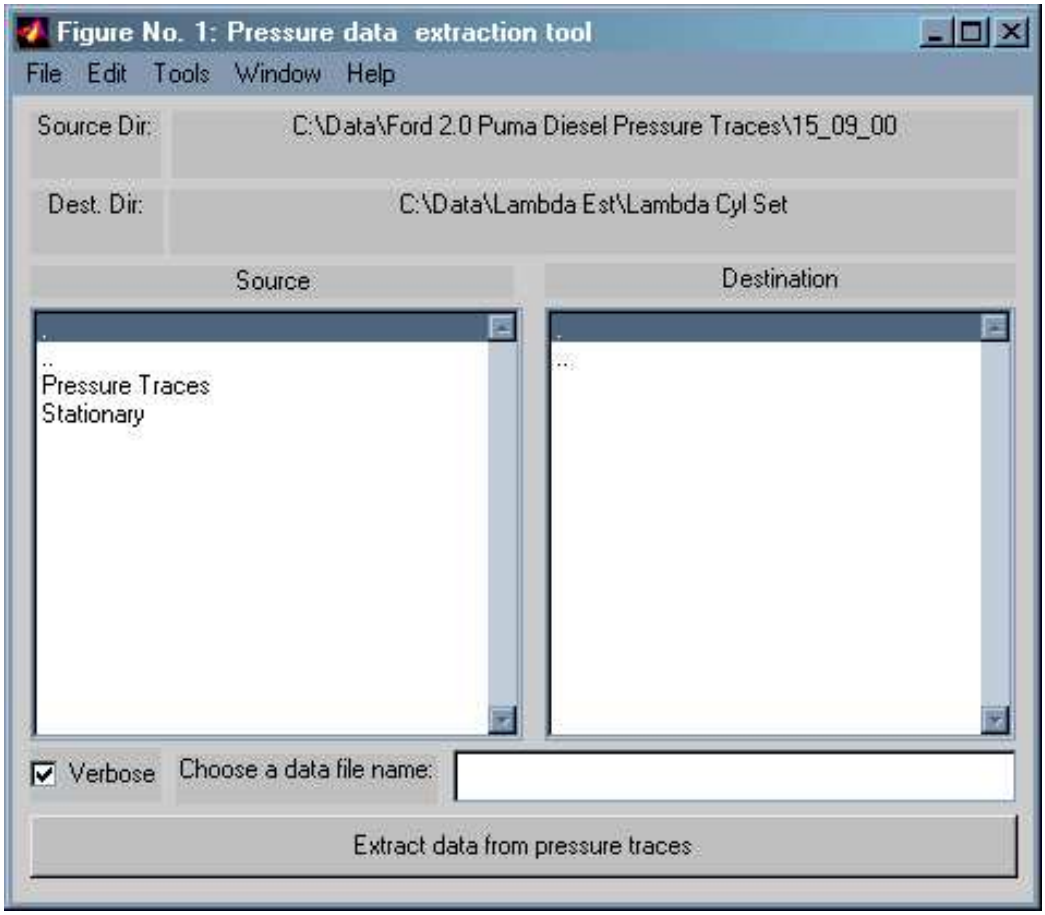


Figure A.2: The Extraction GUI

A.4 Data Conversion Programs

A.4.1 INDISET data conversion

The pressure data is recorded with a system called INDISET at FFA and the data files it is capable of generating cannot immediately be read by MATLAB. So a program was written to convert the data files into .mat files to make it easier to process the data. The source code for the program can be found in the PCON folder on the source code appendix CD coming with this dissertation.

This program is called PCON and figure A.3 shows a screen shot of the GUI.

The options menu contains the following three items.

Figure A.3: PCON Data Converter Program

1. Convert Indiwin ASCII format to MATLAB ASCII format.
2. Convert Indiwin ASCII format to MATLAB MAT file format.
3. Convert Indiwin IFile format to Mat file format.

Which is self explanatory. It is here that one chooses what to convert from and to.

When a conversion type has been chosen, then simply choose open in the file menu and select the files to be converted.

A.4.2 Further development issues

The PCON program can do the conversions, but there are some details that needs to be completed before the program is all done.

These are.

1. The IFile to MAT file conversion has no check on the file operations. So if some files operations for some reason go wrong then the program won't know about it. It will just report that everything went well.
2. The release version does not compile correctly for yet unknown reasons. So one has to use the debug version.

3. There are some memory leaks in the IFile to MAT file conversion method, but it works. These however should be fixed

A.4.3 PUMA data file conversion

The PUMA system generates the stationary measurements in a text file and this text file is not directly readable in MATLAB. A small MATLAB program was written to read this text file and load the data into MATLAB matrices with the same names as in the PUMA text file. The unit names of the data are also stored in a MATLAB matrix called.

- units

The program is called getssdata.m and is part of the extraction.m program that generates data for the neural network training GUI.

A.5 C++ Matrix Library

The C++ matrix library contains the matrix object and several functions necessary to manipulate and do calculations with matrices. The source code for the Matrix Library can be found in the Matrix Control Library/MLib folder on the source code appendix CD coming with this dissertation.

A.5.1 Purpose

The are several purposes with this library.

1. To provide routines to write my own neural network training software.
2. To provide routines for implementing advanced controllers written more easily readable in C++. And more Efficiently.
3. To have my own routines so that I understand them perfectly and are thus able to better suit them to my purposes.

The following is a description of what the library contains and how to use it.

A.5.2 Initialization

The matrices can be initialized (Constructed) in several ways as can be seen in table A.2.

Matrix Constructors		
Matrix	A;	Constructs an empty matrix with no name.
Matrix	A("","Ui")	Constructs an empty matrix with the name: Ui
Matrix	A(m,n)	Constructs an empty matrix with m rows, n columns and no name.
Matrix	A(m,n,"Ui")	Constructs an empty matrix with m rows, n columns and with the name "Ui".
Matrix	A("1,2;3,4","Ui")	Constructs this matrix: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ with the name Ui.
Matrix	A(B)	Constructs a copy of the matrix B.
Matrix	A(m,n,data)	Constructs a matrix with m rows, n columns, filling it with data from the real array data.
Matrix	A(m,n,data,"Ui")	Constructs a matrix with m rows, n columns and the name Ui, filling it with data from the real array data.
	A="1,2;3,4"	Initializes the already constructed matrix A to be $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ Deleting the old values in A.

Table A.2: Matrix Constructors.

A.5.3 Matrix operators

Binary and unary operators

The matrix library is written in C++ and uses the operator overloading features of C++. This makes it possible to write matrix formula as in MATLAB, but in your C++ program. The data type real is just a `#DEFINE` statement and can be changed to any data type that one wants the matrix elements to be.

The matrix class contains the operators listed in table A.3.

External operators

Some operators were defined outside the Matrix class to make operations like <real> + <Matrix> possible. Those operators are described in table A.4

A.5.4 Matrix functions

Information functions in the Matrix class

There are some information functions build into the Matrix class as member functions. They are listed in table A.5.

Information functions outside the class

The matrix library also contains a set of mathematical and utility functions to assist in writing matrix based algorithms.

These functions are listed in tables A.6, A.7, A.8 and A.9.

Matrix Operators			
<Type>	Operator	<Type>	Description
<Matrix>	+	<Matrix>	Adds two matrices
<Matrix>	+	<real>	Adds the real scalar to all the matrix's elements
<Matrix>	+	<int>	Adds the integer scalar to all the matrix's elements
	-	<Matrix>	Changes the sign on all the matrix's elements
<Matrix>	-	<Matrix>	Subtracts two matrices
<Matrix>	-	<real>	Subtracts the real scalar from all the matrix's elements
<Matrix>	-	<int>	Subtracts the integer scalar from all the matrix's elements
<Matrix>	*	<Matrix>	Multiplies two matrices
<Matrix>	*	<real>	Multiplies the real scalar with all the matrix's elements
<Matrix>	*	<int>	Multiplies the integer scalar with all the matrix's elements
<Matrix>	/	<real>	Divides the elements of the matrix with the real scalar
<Matrix>	/	<Matrix>	Internal division of two matrices.
<Matrix>	=	<Matrix>	Assigns a matrix to a matrix variable. New memory is not allocated if there is already enough. Same for all = ops.
<Matrix>	=	<real>	Assigns a real to a matrix and thus making that matrix a 1 x 1 matrix.
<Matrix>	=	<char*>	Initializes the matrix with the values in the string pointed to by the <char*> (See section A.5.2.
<Matrix>	%	<Matrix>	Internal product on the two matrices.
<Matrix>	&	<Matrix>	Concatenates the two matrices vertically
<Matrix>	|	<Matrix>	Concatenates the two matrices horizontally
<Matrix>	(int r, int c)		Returns the element at row r and column c
<Matrix>(Matrix r, Matrix c)			Returns a matrix containing the elements at the rows and columns specified in the matrices r and c

Table A.3: Matrix Operators.

Matrix External Operators			
<Type>	Operator	<Type>	Description
<real>	+	<Matrix>	Adds the real scalar to all the matrix's elements
<real>	-	<Matrix>	Returns a matrix where each element is the real minus the corresponding element in the matrix.
<real>	*	<Matrix>	Multiplies the real scalar with all the matrix's elements
<real>	/	<Matrix>	Returns a matrix where each element is the real divided by the corresponding element in the matrix.

Table A.4: Matrix External Operators.

Matrix Member Functions		
<R. Type>	<Member function>	Description
<void>	Print(int d, int p)	Prints the matrix to the console with d digits and p precision.
<void>	Trace(int d, int p)	Prints the matrix to the debug console in Microsoft Visual Studio with d digits and p precision.
<void>	MsgBox(int d, int p)	Prints the matrix in a Message Box with d digits and p precision.
<void>	Reallocate(int r, int c, bool erase)	Allocates enough memory for an $r \times c$ matrix if necessary.
<void>	ReadArray(int r, int c, real* init, char* name, bool byrows)	Makes the matrix an $r \times c$ with $r \times c$ values taken row wise (if byrows=true) from the array pointed to by init.
<int>	GetCols()	Returns the number of columns in the matrix.
<char*>	GetName()	Returns a pointer to the Matrix name, NULL if no name has been given.
<int>	GetRows()	Returns the number of rows in the matrix.
<real*>	GetData()	Returns a pointer to the matrix elements.
<void>	SetName(char* n)	Sets the Matrix name to n.

Table A.5: Matrix Member Functions.

Row and Column Operations		
<R. Type>	Function	Description
<void>	ColAdd(Matrix &A, int col, real av)	Adds av to all the elements in column col in the matrix A.
<void>	ColAdd(Matrix &A, Matrix &av)	Adds the first element of av to all the elements in the first row in the matrix A and so on.
<void>	ColDiv(Matrix &A, Matrix &dv)	Divides the columns in the matrix A with values in dv. First row in dv with first row in A.
<void>	ColOpAdd(Matrix &A, int c1, int c2, real f)	Adds column $c2 \cdot f$ to column c1 in matrix A.
<void>	ColProduct(Matrix &A, Matrix &mv)	Multiplies the columns in the matrix A with values in mv. First value in mv with first column in A.
<void>	RowAdd(Matrix &A, int row, real av)	Adds av to all the elements in row row in the matrix A.
<void>	RowAdd(Matrix &A, Matrix &av)	Adds the first element of av to all the elements in the first column in the matrix A and so on.
<void>	RowDiv(Matrix &A, int r, real dv)	Divides row r in matrix A by dv.
<void>	RowMul(Matrix &A, Matrix &mv)	Multiplies the rows in the matrix A with values in mv. First value in mv with first row in A. Etc.
<void>	RowMul(Matrix &A, int r, real mv)	Multiplies row r in matrix A by mv.
<void>	RowOpAdd(Matrix &A, int r1, int r1, real f)	Adds row $r2 \cdot f$ to row r1 in matrix A.
<void>	ShiftColumnsLeft(Matrix &A, Matrix &si)	Shifts the columns in A to the left and puts the vector si in the last column.
<void>	ShiftColumnsRight(Matrix &A, Matrix &si)	Shifts the columns in A to the right and puts the vector si in the first column.
<void>	ShiftLeft(Matrix &A, real shiftin)	Shifts all elements in A to the left. Leftmost values in a row becomes the first in the row above. shiftin becomes the last element in A.
<void>	ShiftRight(Matrix &A, real shiftin)	Shifts all elements in A to the right. Rightmost values in a row becomes the first in the row below. shiftin becomes the first element in A.
<void>	SolveBackwardsUp(Matrix &A)	Performs Gauss operations on A until the left part of A is a unity matrix.
<void>	UpperTriangle(Matrix &A)	Performs row operations on A to make it upper triangle.

Table A.6: Row and Column Operations.

Utility functions		
<R. Type>	Function	Description
<void>	Clear(Matrix &A)	Sets the row and column number to zero without freeing memory.
<void>	ConvertToVec(Matrix &A)	Sets the number of rows to the number of elements in A and the number of columns to 1.
<void>	ConvertToRowVec(Matrix &A)	Sets the number of columns to the number of elements in A and the number of rows to 1.
<void>	DiagAdd(Matrix &A, real av)	Adds av to all the diagonal elements in. the matrix A.
<Matrix>	Eye(int n)	Returns an n x n unit $r \times c$ matrix.
<Matrix>	ExtractMatrix(Matrix &A, int rp, int cp,int r, int c)	Returns the $r \times c$ matrix at row rp, column cp in A
<Matrix>	ExpandMatrix(Matrix &A, int rows, int cols)	Returns a block matrix, one block for each element in A, where each $rows \times cols$ block is a matrix consisting of elements equal to the corresponding element in A.
<void>	ExtendMatrixVertically(Matrix &A, Matrix &B)	Same as the operator &, but it utilizes A's memory if there is enough otherwise a new memory block is allocated.
<Matrix>	GrowMatrix(Matrix &A, int rows, int cols)	Returns a $rows \times cols$ block matrix Where each block is a copy of A.
<Matrix>	Mat2Vec(Matrix &A, bool rowwise)	Returns an $n \times 1$ matrix, elements taken row wise from the $r \times c = n$ matrix A. rowwise=false $\Rightarrow$ column wise
<Matrix>	Ones(int r, int c)	Returns an $r \times c$ all ones matrix
<void>	PutColumn(Matrix &A, int c, const Matrix &cv)	Overwrites column c in matrix A taking elements row wise from matrix cv.
<Matrix>	Randm(int r, int c)	Returns a random $r \times c$ Matrix.
<void>	Reshape(Matrix &A, int r, int c)	Sets the number of rows and columns to r and c.
<void>	Step(Matrix &A, int pos, real sv)	Adds sv to element #pos in Matrix A.
<Matrix>	SubMatrix(Matrix &A, int Row, int Col)	Takes out row Row and column Col from matrix A and returns the resulting matrix.
<Matrix>	Vec2Mat(Matrix &A, int pos, int r, int c)	Returns an $r \times c$ matrix of elements taken row wise from the matrix A. (Often an $n \times 1$ matrix)
<void>	Zero(Matrix &A, int rp, int cp, int r, int c)	Inserts zero elements in Matrix A from row rp and column cp to row rp+r and column cp+c.
<Matrix>	Zeros(int r, int c)	Returns an $r \times c$ zero matrix

Table A.7: Matrix Utility Functions.

Matrix Information Functions		
<R. Type>	Function	Description
<int>	Length(Matrix &A)	Returns the largest value of the number of columns or rows.

Table A.8: Matrix Information Functions.

Mathematical Matrix Functions		
<R. Type>	Function	Description
<Matrix>	atan(Matrix &A)	Returns a matrix where each element is atan of the corresponding element in A.
<Matrix>	ColSum(Matrix &A)	Adds all the columns together and returns a column vector with the sums.
<Matrix>	ColSums(Matrix &A, int n)	Adds all the columns together and returns a matrix with the sums.
<real>	Det(Matrix &A)	Returns the determinant of the matrix A.
<real>	DiagProd(Matrix &A)	Returns the product of all the diagonal elements in A.
<real>	DiagSum(Matrix &A)	Returns the sum of all the diagonal elements in A.
<Matrix>	Inv(Matrix &A)	Returns the inverse of the matrix A.
<real>	Norm(Matrix &A)	Returns the square root of the sum of all the elements in A squared.
<Matrix>	RowSum(Matrix &A)	Adds all the rows together and returns a row vector with the sums.
<Matrix>	Solve(Matrix &A, Matrix &B)	Solves the equation $Ax = B$.
<real>	SqrSum(Matrix &A)	Returns the square sum of the elements in A.
<real>	Sum(Matrix &A)	Returns the sum of all the elements in the matrix A.
<Matrix>	tanh(Matrix &A)	Returns a matrix where each element is tanh of the corresponding element in A.
<Matrix>	Transpose(Matrix &A)	Returns the transpose of the matrix A.

Table A.9: Mathematical Matrix Functions.

A.5.5 Examples

This section contains some examples of how to create and initialize matrices, perform matrix arithmetics and using mathematical functions on the matrices.

Initialization

To create an empty matrix:

```
Matrix A;
```

To create an empty matrix with a name:

```
Matrix A(0,0,"SuperMatrix1");
```

The $0, 0$ parameters only means that the matrix is empty. If one wants an uninitialized matrix with a name, then it looks like this:

```
Matrix A(3,4,"SuperMatrix2");
```

This matrix is then an uninitialized 3 x 4 matrix with the name $SuperMatrix2$. The elements in the matrix are whatever there was in the memory allocated at the time.

To create a preinitialized matrix with an optional name:

```
Matrix A("1,2;3,4","SuperMatrix3");
```

This creates the following 2 x 2 matrix: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

To create a 1 x 1 matrix from a real number:

```
Matrix A(2);
```

This one contains the number 2.

To create a matrix from an array of reals:

```
real   data={1.1, 2.2, 3.3, 4.4};
Matrix  A(2, 2, data);
```

Matrix arithmetics

Because of the operator overloading features in C++, it is possible to perform additions, multiplications and so on as you would with a double or an integer. So a simple random matrix formula should suffice to demonstrate the use of matrices in formulas.

```
Matrix                  A("1,2;3,4");
Matrix                  B,C,D,E,F;
real                    d;

B=A*A*A+2*A*A-A;
C=B+2;
D=Inv(A);
d=Det(A);
E=(A | B) & (B | A);
F=Solve(A,B);
```

This calculates the matrix polynomial $A^3 + 2A^2 - A$ and puts the result in the matrix B.

The second formula adds the real number 2 to all the elements in B and puts the result in the matrix C.

The next two lines demonstrates the use of two mathematical functions on the matrix A. The inverse and the determinant.

Then a demonstration of the concatenation operators | and &. A and B are concatenated vertically(|) both ways and the two vertical concatenations are then concatenated horizontally(&).

The last line is an example of how to solve a linear set of equations.

$$Ax = B \tag{A.2}$$

A.5.6 No checking of any kind in the library!

The matrix library contains no error checking whatsoever to make it as fast as possible. So it will crash if one for instance indexes a matrix out of range.

A.6 Optimization Library

A.6.1 Purpose

The purpose of developing this optimization library is to be able to implement C++ versions of neural network training classes and to write a C++ version of a neural predictive controller that can be used for online control. The source code for the Optimization Library can be found in the Matrix Control Library/OptimizerLib folder on the source code appendix CD coming with this dissertation.

A.6.2 How to Use the Library

The library has one base optimization (Optimizer) class to serve as the foundation for future optimization classes and a single Levenberg-Marquardt optimization class (Marquardt) which is a specialization of the Optimizer class. The Optimizer class is an empty class and cannot be utilized. It is only a base class which will be made abstract in the future.

A more in depth description of the Levenberg-Marquardt optimization algorithm can be found in the original paper by Marquardt in [27] and in this dissertation in section 5.1.4.

The parameters for the Marquardt class is listed and described in table A.10. Basically it is used by creating an instance of the class with one of the constructors and provide it with the necessary optimization parameters such as the stop criterion, maximum number of iterations and pointers to the error functions that should be minimized.

The main `Minimize(X0` method is then called with the starting parameter vector guess `X0` which performs the actual optimization and returns the result.

The Marquardt class minimizes the following cost function.

$$e = \frac{1}{2}\sum_{i=1}^{fn} Mat2Vec(f_i(x, userdata))^T Mat2Vec(f_i(x, userdata)) \qquad \text{(A.3)}$$

The functions f_i are usually error functions and x is a vector containing the parameters for the functions that the error function will be minimized with respect to.

Marquardt Class Optimization Parameters	
Stop Norm	The optimization will stop if the norm of the gradient (5.25) of the cost function (5.23) is below this value.
Delta	This is the step value for the numerically calculated jacobian if the jacobian is not provided.
Initial Lambda Value	The initial value for the dampening parameter.
Maximum Iterations	The optimization will stop if the number of iterations reach this value.
Number of f_is	The number of error functions to be minimized.
Pointer to an array of f_is	An array of pointers to the error functions.
Pointer to an array of $f_i^{'}$s	An array of optional pointers to the derivatives of the error functions. Each entry in the array can be made NULL if a numerically calculated derivative is desired.
Number of free parameters	The number of free parameters to be found by optimization.
Pointer to userdata	A pointer to extra information needed by the error functions. This pointer will be a passed to the error functions along with the parameter to be found.

Table A.10: Marquardt Class Optimization Parameters

$Userdata$ is a pointer to other kinds of data needed by the functions such as physical constants.

Notice the Mat2Vec (See table A.7) function is used on the output of these functions. It is done to make it possible to handle several input sets to these functions simultaneously by organizing the outputs of the f_i functions in columns, where each column represents the output vector corresponding to an input in a column in the x matrix.

The member functions in the Marquardt class shown in table A.11 is used to specify the f_i functions and the optimization parameters.

It is also possible to specify derivatives of the f_i functions for better accuracy and possibly also better speed, otherwise a numerical derivative will be calculated.

A.6.3 Marquardt Member Functions

The member functions of the Marquardt class are listed and explained in table A.11.

Marquardt Member Functions		
<R. type>	<Member function>	Description
	Marquardt()	Constructs a Marquardt optimizer instance. class with all the default parameters.
	Marquardt(fptr *cf, void* ud=NULL, int fn=0, real sn=1e-03, int mi=100)	This constructs a marquardt optimizer instance where one has the possibility to specify some of the parameters for the optimization. cf is a pointer to an array of f_is. ud is a pointer to the userdata. It's a void pointer so that it's possible to point to anything(A matrix or an object). fn is the number of f_is. sn is the stop norm of the gradient of the error function A.3. Optimization stops when the norm of that gradient is below sn. mi is the max. number of iterations allowed.
<real>	GetDelta()	Returns the delta value used to calculate the numerical derivative.
<fptr*>	GetFFunctions()	Returns a pointer to the array of f_is.
<int>	GetFN()	Returns the number of f_is.
<int>	GetIter()	Returns the number of iterations used.
<fptr*>	GetJFunctions()	Returns a pointer to the array of derivative functions.
<real>	GetLambda()	Returns the value of the Marquardt lambda parameter.
<int>	GetMaxIter()	Returns the maximum number of iterations allowed.
<int>	GetN()	Returns the number of parameters that A.3 is being minimized with respect to.
<real>	GetStopNorm()	Returns the stop norm.
<void*>	GetUserData()	Returns the userdata pointer.
<Matrix>	Minimize(Matrix X0)	Starts the minimization and returns the result.
<void>	SetDelta(real const &dv)	Sets the delta value.
<void>	SetFFunctions(fptr* ff)	Sets the pointer to the array of f_is.
<void>	SetFN(int const &nf)	Sets the number of f_is.
<void>	SetJFunctions(fptr* jf)	Sets the pointer to the array of derivative functions.
<void>	SetLambda(real const &lv)	Sets the lambda value.
<void>	SetMaxIter(int const &miv)	Sets the maximum number of iter. allowed.
<void>	SetStopNorm(real const &snv)	Sets the stop norm value.
<void>	SetUserData(void* ud)	Sets the userdata pointer.

Table A.11: Marquardt Member Functions.

The fptr type used in the Marquardt class member function table is a pointer to a static function of the following type.

```
Matrix f(Matrix &A, void* userdata)
```
(A.4)

Notice the `void*` pointer which is used to transfer all other required data for the function. The error functions f_i and their optional derivative functions must be of this type.

This parameter could for instance be used to transfer the pointer to a C++ class containing all the data necessary for the function. Possibly the same class containing the error functions f_i.

The default optimization parameters for the Marquardt class are listed in table A.12.

Marquardt Default Parameters	
Parameter	Default value
Stop Norm	1.0e-03
Delta	1.0e-06
Initial Lambda Value	1.0
Maximum Iterations allowed	100
Number of f_is	0
Pointer to the array of f_is	NULL
Pointer to the array of derivative functions	NULL
Number of parameters to be found	0
Pointer to userdata	NULL

Table A.12: Marquardt Default Parameters

A.6.4 Derivative Functions.

The derivative functions are of the same type as the f_is (See A.4).

If the pointer to the array of derivative functions is NULL then a numerical derivative will be calculated using a step length of Delta.

If the array of pointers to derivative functions contains NULL values then those derivatives will be calculated numerically using the step length Delta.

A.6.5 Example

Here is an example of how to use the optimization library. In this example the derivative functions are not specified and they will therefore be calculated numerically.

```
#include "../mlib/Matrix.h"
#include "Marquardt.h"

Matrix  x0  = "4;5;6";
Matrix  A   = "5,2,8;9,2,1;3,6,3";

Matrix  f(Matrix x, void* ud)
{
    return A * ( (x%x) - (*((double*)ud)) * x0 );
}

Matrix nextf(Matrix x, void* ud)
{
    return 0.3 * x;
}

int main(int argc, char* argv[])
{

    Marquardt::fptr     flist[2];
    double              constant=2.0;

    flist[0]    = f;
    flist[1]    = nextf;

    Marquardt   Opti(flist, &constant, 2, 1e-8, 25);
    Matrix      Result;
    Matrix      X0("1;2;3");

    printf("Optimization Test!\n");
```

```
    Result=Opti.Minimize(X0);

    printf("Result =\n");
    Result.Print(5,4);

    printf("Iterations = %d\n",Opti.GetIter());

    return 0;
}
```

A.7 Neural Network Library

The neural network library contains a base neural network class (`NeuralNetwork`) and a single specialization of it. This specialization is a single hidden layer neural network (See section 1.3) called `SHLNetwork`.

The library also contains two neural network training classes called `NetworkTrainer` and PredictiveNetworkTrainer.

The source code for the Neural Network Library can be found in the Matrix Control Library/NetworkTrainer folder on the source code appendix CD coming with this dissertation. Some MATLAB MEX functions for neural network training from the MATLAB commando prompt has also been made and the source code for those can be found in the folder ToolBoxes/Mex.

The base class `NeuralNetwork` is basically an empty class that merely defines the basic member functions of a neural network. Its member functions are empty and should be overridden by specializations. The `SHLNetwork` class overrides all the member functions of the `NeuralNetwork` class and the description of both classes is thus combined in the description of the `SHLNetwork` class. The formats of the various input and output matrices for the `SHLNetwork` class are however specific to that specialization and future specializations will not necessarily utilize the same format.

It is the intention that the `NeuralNetwork` class should be a pure abstract class in the future.

A.7.1 The Single Hidden Layer Network Class

The class name is SHLNetwork.

This class contains the necessary member functions to calculate the output and the derivative of a single hidden layer neural network(SHLNetwork). And of course functions to set and get the weights and biases of the network.

Input-Output Sets

The SHLNetwork class can handle several input sets at the same time. An input set is one set of inputs that belongs to a specific set of outputs. An example.

X	Y	Z	W
1	1	1	2
1	2	2	3
2	1	2	3
2	2	4	4

Table A.13: Input-output set example.

Table A.13 is an example of 4 input-output sets. $X = 1$, $Y = 1$, $Z = 1$ and $W = 2$ is one set and these numbers belong together. $X = 1$, $Y = 1$ is the input set in that input-output set. and $Z = 1$, $W = 2$ is the output set.

The input matrix to the function in the SHLNetwork class that calculates the output of the network has a special format to store several input sets in it. The output matrix also has a similar special format.

The various matrix formats and functions used in the SHLNetwork class is explained in the following sections.

Input Matrix Format

The network can have more than one input and it can calculate the outputs corresponding to several input sets simultaneously.

The following matrix shows how to construct the input matrix to the SHLNetwork class.

$$\begin{bmatrix} x_{11} & \cdots & x_{1n_s} \\ \vdots & & \vdots \\ x_{n_{in}1} & \cdots & x_{n_{in}n_s} \end{bmatrix} \tag{A.5}$$

x_{ij} Is input number i belonging to input output set j.
n_s Is the number of input sets.
n_{in} Is the number of inputs.

Output Matrix Format

The output matrix is formatted in a similar way as the input matrix.

$$\begin{bmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & & \vdots \\ y_{n_{out}1} & \cdots & y_{n_{out}n_s} \end{bmatrix} \tag{A.6}$$

y_{ij} Is output number i belonging to input output set j.
n_s Is the number of input sets.
n_{out} Is the number of outputs.

Weight Matrices Format

The SHLNetwork stores the weights and biases in 4 matrices.

- The hidden layer weights matrix, W_1.
  These weights are the ones that are multiplied with the inputs. The sum of these multiplications and the hidden biases is then given as input to the activation function of the neurons. Each row in this matrix corresponds to a neuron.

- The hidden layer biases, B_1.
  See the explanation for the hidden layer weights matrix. Each row in this vector corresponds to a neuron.

- The output layer weights matrix, W_2.
  These weights are the ones that are multiplied with the outputs of the neurons activation functions. The sum of these multiplications and the output biases forms the outputs of the neural network. Each row in this matrix corresponds to an output of the network.

- The output layer bias, B_2.
  See the explanation for the output layer weights matrix. Each row in this vector corresponds to an output.

These matrices can be set directly using the member function SetWeights() (See A.14).

The matrices looks like this.

Hidden layer weights.

$$W_1 = \begin{bmatrix} w_{111)} & \cdots & w_{11n_{in}} \\ \vdots & \ddots & \vdots \\ \vdots & w_{1ij} & \vdots \\ \vdots & \ddots & \vdots \\ w_{1n_h1)} & \cdots & w_{1n_hn_{in}} \end{bmatrix} \tag{A.7}$$

Hidden layer biases.

$$B_1 = \begin{bmatrix} b_{11} \\ \vdots \\ b_{1i} \\ \vdots \\ b_{1n_h} \end{bmatrix} \tag{A.8}$$

Output layer weights.

$$W_2 = \begin{bmatrix} w_{211} & \cdots\cdots\cdots & w_{21n_h} \\ \vdots & \ddots & \vdots \\ \vdots & w_{2ij} & \vdots \\ \vdots & \ddots & \vdots \\ w_{2n_{out}1} & \cdots\cdots\cdots & w_{2n_{out}n_h} \end{bmatrix} \tag{A.9}$$

Output layer biases.

$$B_2 = \begin{bmatrix} b_{21} \\ \vdots \\ b_{2i} \\ \vdots \\ b_{2n_{out}} \end{bmatrix} \tag{A.10}$$

Where

n_{in}	The total number of inputs to the network (in each set).
n_{out}	The total number of outputs of the network (in each set).
n_h	The total number of hidden neurons in the network.
w_{1ij}	is a hidden layer weight in the i'th hidden layer neuron for the j'th input.
b_{1i}	is the bias (offset) in the i'th hidden layer neuron.
w_{2ij}	is the output layer weight in the j'th output neuron for the i'th output.
b_{2i}	is the bias (offset) in the i'th output neuron.

Weight Vector Data Format

The weights and biases of the SHLNetwork class can be set by using a single vector. This is useful for algorithms like the Marquardt optimization that needs to get a derivative with respect to a vector.

The weights and biases can be retrieved and set as a single vector containing the parameters for both the hidden layer and the output layer.

The function

```
SetWeights(Matrix &W)
```

utilizes this format for the parameter matrix W and the function

```
Matrix GetWeights()
```

returns a matrix in this format. The vector format has the following structure.

$$W_{wv} = Mat2Vec \left(\begin{bmatrix} w_{111} & \cdots & w_{11n_{in}} \\ w_{121} & \cdots & w_{12n_{in}} \\ w_{1n_h1} & \cdots & w_{1ij} \\ b_{11} & \cdots & b_{1n_h} \\ w_{211} & \cdots & w_{21n_h} \\ w_{221} & \cdots & w_{22n_h} \\ w_{2n_{out}1} & \cdots & w_{2n_{out}n_{in}} \\ b_{21} & \cdots & b_{2n_{out}} \end{bmatrix} \right) \tag{A.11}$$

Basically the `GetWeights()` function is just taking the the elements row wise from the matrices W_1,B_1,W_2 and B_2 in that order.

Order of the Derivative Outputs Elements

The SHLNetwork class contains methods to calculate the derivative of the neural network outputs with respect to both the inputs and the weights. These methods are used to provide the necessary information for predictive control (inputs) and training (weights).

For the derivative with respect to the weights:

The elements in the derivative output from the SHLNetwork class is ordered in a special way to make it as easy as possible to handle for an optimization routine.

The order of the elements in the derivative matrix is based on the order of elements in the weight vector format(See A.7.1).

Row i and column j in the derivative vector is the partial derivative of network output $int(\frac{i}{n+1}) + 1$ with respect to element j in the weight vector(See A.11). Where n is the number of input sets.

It looks like this.

$$\begin{bmatrix} \frac{\partial y_{11}}{\partial w_{wv_1}} & \cdots & \frac{\partial y_{11}}{\partial w_{wv_{nwv}}} \\ \vdots & & \vdots \\ \frac{\partial y_{1n}}{\partial w_{wv_1}} & \cdots & \frac{\partial y_{1n}}{\partial w_{wv_{nwv}}} \\ \frac{\partial y_{2n}}{\partial w_{wv_1}} & \cdots & \frac{\partial y_{2n}}{\partial w_{wv_{nwv}}} \\ \vdots & & \vdots \\ \frac{\partial y_{2n}}{\partial w_{wv_1}} & \cdots & \frac{\partial y_{2n}}{\partial w_{wv_{nwv}}} \\ \vdots & & \vdots \\ \frac{\partial y_{mn}}{\partial w_{wv_1}} & \cdots & \frac{\partial y_{mn}}{\partial w_{wv_{nwv}}} \end{bmatrix} \tag{A.12}$$

For the derivative with respect to the inputs:

$$\begin{bmatrix} \frac{\partial y_{11}}{\partial x_1} & \cdots & \frac{\partial y_{11}}{\partial x_{n_{in}}} \\ \vdots & & \vdots \\ \frac{\partial y_{1n}}{\partial x_1} & \cdots & \frac{\partial y_{1n}}{\partial x_{n_{in}}} \\ \frac{\partial y_{21}}{\partial x_1} & \cdots & \frac{\partial y_{21}}{\partial x_{n_{in}}} \\ \vdots & & \vdots \\ \frac{\partial y_{2n}}{\partial x_1} & \cdots & \frac{\partial y_{2n}}{\partial x_{n_{in}}} \\ \vdots & & \vdots \\ \frac{\partial y_{mn}}{\partial x_1} & \cdots & \frac{\partial y_{mn}}{\partial x_{n_{in}}} \end{bmatrix} \tag{A.13}$$

y_{ij}	is the output i corresponding to the input set j.
w_{wv_i}	is the element i from the weight vector (See A.11).
n	is the number of input sets.
m	is the number of outputs.
nwv	is the number of elements in the weight vector.

SHLNetwork member functions

The functions of the SHLNetwork class are listed and explained in the table A.14.

SHLNetwork Member Functions		
<R. Type>	Function	Description
	SHLNetwork()	Constructs an instance of the SHLNetwork class using the default set of parameters.
	SHLNetwork(int hn, int in, int out)	Constructs an instance of the SHLNetwork class with hn neurons in the hidden layer, in inputs and out outputs. The network weights and biases will be initialized with random values.
	SHLNetwork(Matrix hw, Matrix hb, Matrix ow, Matrix ob)	Constructs an instance of the SHLNetwork class using the network weights and biases specified in hw : Hidden layer weights. hb : Hidden layer biases. ow : Linear output layer weights. ob : Linear output layer biases. (See A.7.1)
<Matrix>	GetWeights()	Returns the Network weights and biases in the format specified in A.11.
	InitNet()	Assigns random values to all the weights and biases.
<Matrix>	NetworkDerivative(Matrix &Input)	Returns the derivative of the SHL Networks output with respect to the weights and biases. The order of the elements are shown in A.12.
<Matrix>	NetworkDInputs(Matrix &Input)	Returns the derivative with respect to the inputs.
<Matrix>	NetworkOut(Matrix &Input)	Returns the output of the SHL Network given the input Input.
<int>	NumberOfWeights()	Returns the number of weights in the Network.
<void>	SetWeights(Matrix &W)	Sets the weights and biases of network. W is a vector packed in format specified in A.11
<void>	SetWeights(Matrix &W1, Matrix &W2)	Sets the weights and biases. $W1$ are the hidden weights with the hidden biases in the last column. $W2$ are the output weights with the output biases in the last column.
<void>	SetWeights(Matrix &hw, Matrix &hb, Matrix &ow, Matrix &ob)	Sets the weights and biases as in the matching constructor above. The data format for these matrices can be seen in A.7.1.

Table A.14: SHLNetwork Member Functions

SHLNetwork Default Values

The default values in the SHLNetwork class are listed in table A.15

SHLNetwork Default Parameters	
Parameter	Default value
Number of neurons in the hidden layer.	5
Number of inputs.	1
Number of output.	1
Weights and biases.	Random
Name	""

Table A.15: SHLNetwork Default Parameters

A.7.2 The Network Trainer Class

The neural network library also contains a network training class called NetworkTrainer.

This class uses the Marquardt class to find the weights and biases that makes the network produce output sets as close as possible to some desired output sets given certain input sets.

It simply works by giving it a pointer to a NeuralNetwork based class like the SHLNetwork class and then set the various optimization parameters such as the maximum number of iterations, stop criterion and decay weight with the appropriate Set member functions. Then all the input and target output data must be given using SetNetInData() and SetNetOutData().

When all the necessary information has been given, then a call to the Train() method can be executed in order to train the network.

Table A.16 contains a list of all the member functions in the NetworkTrainer class and a description of their inputs and outputs.

NetworkTrainer Member Functions

The table A.16 contains the member functions and their explanations used to setup a neural network training run.

NetworkTrainer Member Functions		
<R. Type>	Function	Description
	NetworkTrainer()	Constructs an instance of the NetworkTrainer class using the default parameters.
	NetworkTrainer(NeuralNetwork* nn, int maxi, double stopn, double dec)	Contructs an instance of the NetworkTrainer class with: nn : The neural network to be trained (f.ex. an SHLNetwork). maxi : Max. nr. of iterations allowed for training. stopn : The stop norm utilized for training. dec : The decay value (See equation 1.3).
<real>	GetDecay()	Returns the decay value (See equation 1.3).
<Matrix>	GetNetInData()	Returns the training input data.
<Matrix>	GetNetOutData()	Returns the desired output data.
<int>	GetMaxIter()	Returns the maximum number of training iterations allowed.
<real>	GetStopNorm()	Returns the stop norm value (See table A.11).
<NeuralNetwork*>	GetTheNetwork()	Returns a pointer to the specified neural network.
<void>	SetDecay(real dv)	Sets the decay value (See 1.3).
<void>	SetNetInData(Matrix &id)	Sets the input data matrix to id.
<void>	SetNetOutData(Matrix &od)	Sets the target output data matrix to od.
<void>	SetMaxIter(int mi)	Sets the maximum iterations value.
<void>	SetStopNorm(real sn)	Sets the stop norm value. (See A.11).
<void>	SetTheNetwork(NeuralNetwork* nn)	Sets the pointer to the neural network to train.
void	SetVerbose(bool v)	Print optimization info or not.
void	SetMatlab(bool m)	Enables or disables updating of the optimization info in the Matlab window
<void>	Train()	Starts the training of the specified neural network with the chosen parameters.

Table A.16: NetworkTrainer Member Functions

The Training Input and Output Data Format

The NetworkTrainer class can train a multi input multi output neural network to fit an entire data set in one training run. The input data and matching output data thus has to be formatted in the correct way for proper processing.

The training data is given to the class through the

```
SetNetInData()
```

and the

```
SetNetOutData()
```

methods and is formatted just as for the network input and output data in the SHLNetwork class in equation A.5 and A.6.

A.7.3 NetworkTrainer Default Parameters

The default training parameters are applied when the empty `NetworkTrainer()` constructor is called. Those parameters are listed in table A.17.

NetworkTrainer Default Parameters	
Parameter	Default value
Network pointer	NULL
Maximum number of iterations	100
Stop Norm (See the Marquardt class in section 5.1.4	0.001
Weights and biases.	Random
Name	""

Table A.17: SHLNetwork Default Parameters

A.7.4 Example

This section contains a small example of how to use both the SHLNetwork class and the NetworkTrainer class to train the SHLNetwork.

```
#include <stdio.h>
#include "NetworkTrainer/NetworkTrainer.h"
#include "NetworkTrainer/PrunableSHLNetwork.h"
#include "MLib/Matrix.h"
#include "MLib/matfun.h"

int main(int argc, char* argv[])
{
    Matrix indata("1,2,3,4,5;1,2,3,1,2","in");
    Matrix outdata("1,0.5,-1,-2,-3;1,2,3,4,5","out");

    NetworkTrainer  Trainer;
    SHLNetwork      Network(2,2,2);

    printf("Network Training Test!\n");
    printf("Network weights before training=\n");
    Transpose(Network.GetWeights()).Print(2,1);
    printf("Network Output before Training=\n");
    Network.NetworkOut(indata).Print(2,1);

    printf("Training in progress!\n");

    Trainer.SetNetInData(indata);
    Trainer.SetNetOutData(outdata);
    Trainer.SetTheNetwork(&Network);
    Trainer.SetMaxIter(500);
    Trainer.SetStopNorm(1e-4);
    Trainer.SetDecay(0.0);
    Trainer.Train();

    printf("Training completed\n");
    printf("Network weights after training=\n");
    Transpose(Network.GetWeights()).Print(2,1);

    printf("Network output after training=\n");
    Network.NetworkOut(indata).Print(2,1);
    printf("And it’s hopefully close to=\n");
    outdata.Print(2,1);

    return 0;
}
```

A.7.5 The Predictive Network Trainer Class

The class name is PredictiveNetworkTrainer.

The class is the implementation of the predictive neural network training scheme presented in section 3.3. It is utilized in very much the same way as the NetworkTrainer class explained in section A.7.2. The class does however have some extra member functions for the new features such as input and output order specification.

The PredictiveNetworkTrainer class has the same member functions as the NetworkTrainer class listed in table A.16 except for the constructors and some new member functions. The constructors and the new member functions are listed in table A.18.

PredictiveNetworkTrainer Member Functions		
<R. Type>	Function	Description
	PredictiveNetworkTrainer()	Constructs an instance of the NetworkTrainer class using the default parameters.
	PredictiveNetworkTrainer(NeuralNetwork* nn, int *m, int *n, int horizon, int maxi, double stopn, double dec, ErrorTypeType etype)	Constructs an instance of the PredictiveNetworkTrainer using the following parameters. nn : A neural network pointer m : An integer array containing $m_1...m_{n_{in}}$ n : An integer array containing $n_1...n_{n_{out}}$ horizon : The number of steps (ph) into the future in the cost function.
		maxi : Maximum number of iterations. stopn : The Stop Norm. dec : The weight decay value. etype : Always choose absolute. relative is not implemented yet.
<int*>	GetInputOrder()	Retrieves a pointer to the array containing $m_1...m_{n_{in}}$
<int*>	GetOutputOrder()	Retrieves a pointer to the array containing $n_1...n_{n_{in}}$
<void>	SetInputOrder()	Sets a pointer to the array containing $m_1...m_{n_{in}}$
<void>	SetOutputOrder()	Sets a pointer to the array containing $n_1...n_{n_{in}}$
<void>	SetVerbose(bool v)	Print optimization info or not.
<void>	SetMatlab(bool m)	Enables or disables updating of the optimization info in the Matlab window

Table A.18: PredictiveNetworkTrainer Member Functions

A.8 Control Library

A.8.1 Introduction

The C++ Control Library contains the base nonlinear predictive controller class and a neural network specialization of it. The source code can be found in the Matrix Control Library/Controllers folder on the source code appendix CD coming with this dissertation. A MATLAB MEX function for running the neural predictive controller on a SIMULINK model from the MATLAB commando prompt was also made and the source code can be found in the folder ToolBoxes/Mex.

The base predictive controller class is called PredictiveController and requires a pointer to a nonlinear system model function that it will be utilized as the predictor in the predictive control strategy.

The Predictive control strategy is briefly described in section 1.4.4 and a more detailed mathematical description of the algorithm can be found in section 5.1. A brief overview of the strategy is also given in this section.

A.8.2 A Brief Description of Predictive Control Strategy

The C++ function for the system model and the system model derivative function required by the PredictiveController has to have the following fairly general structure.

```
Matrix <functionname>(Matrix &Y, Matrix &U, void* UserData)
```
(A.14)

Representing the following nonlinear system model structure.

$$Y_{k+1}^{v} = F(Y_k^{in}, U_k^{in}) \tag{A.15}$$

Where

$$Y_k^{in} = \begin{bmatrix} y_k^1 & y_{k-1}^1 & \cdots & y_{k-n+1}^1 \\ y_k^2 & y_{k-1}^2 & \cdots & y_{k-n+1}^2 \\ \vdots & \vdots & & \vdots \\ y_k^{n_{out}} & y_{k-1}^{n_{out}} & \cdots & y_{k-n+1}^{n_{out}} \end{bmatrix} = \begin{bmatrix} Y_k^v & Y_{k-1}^v & \cdots & Y_{k-n+1}^v \end{bmatrix} \quad \text{(A.16)}$$

$$Y_k^v = \begin{bmatrix} y_k^1 & y_k^2 & \cdots & y_k^{n_{out}} \end{bmatrix}^T \quad \text{(A.17)}$$

$$U_k^{in} = \begin{bmatrix} u_k^1 & u_{k-1}^1 & \cdots & u_{k-m+1}^1 \\ u_k^2 & u_{k-1}^2 & \cdots & u_{k-m+1}^2 \\ \vdots & \vdots & & \vdots \\ u_k^{n_{in}} & u_{k-1}^{n_{in}} & \cdots & u_{k-m+1}^{n_{in}} \end{bmatrix} = \begin{bmatrix} U_k^v & U_{k-1}^v & \cdots & U_{k-m+1}^v \end{bmatrix} \quad \text{(A.18)}$$

$$U_k^v = \begin{bmatrix} u_k^1 & u_k^2 & \cdots & u_k^{n_{in}} \end{bmatrix}^T \quad \text{(A.19)}$$

$F()$	is the nonlinear C^1 system model function.
Y_k^{in}	is the system previous outputs matrix at time k.
Y_k^v	is the system output vector at time k.
U_k^{in}	is the system previous inputs matrix at time k.
U_k^v	is a system input vector at time k.
y_k^i	is system output i at time k.
u_k^i	is system input i at time k.
n	is the number of output samples back in time needed by the model.
m	is the number of input samples back in time needed by the model.
n_{in}	is the number of inputs.
n_{out}	is the number of outputs.

Briefly explained, the predictive controller tries to find the future control signals $U_k, \cdots, U_{k+h_c}$ by minimizing the following cost function.

$$J_k^{npc} = \sum_{i=h_s}^{h_e} (R_{k+i} - Y_{k+i}^v)^T (R_{k+i} - Y_{k+i}^v) + \rho \sum_{i=0}^{h_c} \Delta U_{k+i}^v{}^T \Delta U_{k+i}^v \quad \text{(A.20)}$$
$$\Delta U_k^v = U_k^v - U_{k-1}^v$$

The PredictiveController class then returns the first of the future control signals (U_k^v) as the actual predictive control signal. The member function

```
Matrix Control()
```
(A.21)

performs the optimization of the cost function A.20 and returns the control signal.

The future Y^v_{k+i} in A.20 are predicted using the given system model function in the following way.

$$\begin{array}{lcl} \hat{Y}^v_{k+1} & = & F(Y^{in}_k, U^{in}_k) \\ \hat{Y}^v_{k+2} & = & F(\hat{Y}^{in}_{k+1}, U^{in}_{k+1}) \\ \ldots\ldots & & \end{array} \tag{A.22}$$

The PredictiveController class must also be given a pointer to a function that calculates the derivative of the given nonlinear system model function.

The derivative system model function is the derivative

$$\frac{\partial F}{\partial \begin{bmatrix} Y^v_k \\ U^v_k \end{bmatrix}} \tag{A.23}$$

The UserData parameter is an optional pointer given to the system model function and its derivative function which they can utilize to obtain information necessary to complete the model and derivative calculations.

Typically this will be a pointer to the object in which the model and derivative function is, but can also be a pointer to a data structure.

A.8.3 PredictiveController Class Description

The class name is PredictiveController.

The PredictiveController class is the base class for predictive controllers. There is only one specialization available at this point, the NeuralPredictiveController.

The PredictiveController class can be used as a general nonlinear predictive controller and does not need to be specialized through derivation.

The PredictiveController class is instantiated with the following constructor.

```
PredictiveController(int ni, int no, int yorder,
int uorder, int hs, int he, int hc, double uw)
```
(A.24)

The parameters are explained in table A.19.

PredictiveController Constructor Parameters		
<Type>	Parameter	Description
int	ni	Is the number of inputs (n_{in}) in the system model function.
int	no	Is the number of outputs (n_{out}) in the system model function.
int	yorder	Is the number of previous outputs (n) needed by the system model function.
int	uorder	Is the number of previous inputs (m) needed by the system model function.
int	hs	Is the horizon start (h_s) predictive control parameter. See section 5.1. It is the number of time steps into the future from where the predicted error will be calculated.
int	he	Is the horizon end (h_e) predictive control parameter. See section 5.1. It is the number of time step into the future to where the predicted error will be calculated.
int	hc	Is the control horizon (h_c) predictive control parameter. See section 5.1. It is the number of time steps into the future to where the future control signal will be weighted in the predictive cost function.
int	uw	Is the ρ parameter in the predictive cost function A.20 It weighs the control signal cost against the predicted error.

Table A.19: PredictiveController Constructor Parameters

The parameters for the predictive controller that can be changed in between control signals or merely set before the controller is running are all in the Predictive Control Block (PCB). The PCB can be obtained with by calling the PredictiveController member function.

```
PCData* GetPCB()
```
(A.25)

The members variables in the PCB are listed and explained in table A.20.

The parameters of the PCB must be set before calling the `Control()` member function.

PCB Member Variables		
<Type>	Member	Description
int	MaxIter	The maximum number of iterations allowed to find the optimal future control signals.
double	StopNorm	Optimization stops when the $\mathcal{H}_2$ norm of the cost function gradient is below this value.
double	UWeight	Is the ρ weight in the predictive cost function A.20.
Matrix	UInit	Is the initial past control signals needed by the system model function. See A.18.
Matrix	YInit	Is the initial past system outputs needed by the system model function. See A.16.
Matrix	PresentY	Is a pointer to the current measured outputs.
Matrix	Prediction	This is an informative variable which will keep the prediction of the outputs for the next h_e time steps.
Reference	Matrix	This member must contain the references for the next h_e time steps in a matrix formatted as in 5.19.
See A.14	TheModel	This is the system model function function pointer.
See A.14	TheModelD	This is the system model derivative function.

Table A.20: PCB Members Variables

The UserData parameter in A.14 can be set (if the system model function needs external data) with the member function.

```
void SetTheModelUserData(void* UserData)
```
(A.26)

The member function

```
void Init()
```
(A.27)

must be called after having set all the PCB parameters since some initializations has to be performed before the controller is activated.

The controller is now ready and the member function

```
Control()
```
(A.28)

can be called in order to calculate the predictive control signal.

This can be repeated at every sample time without changing anything since the PredictiveController object keeps track of all necessary information itself.

The table A.21 contains all the PredictiveController class's member functions for quick reference.

Predictive Controller Member Functions		
<R. Type>	Function	Description
Matrix	Control()	Calculates the predictive control signal based on the given model function, its derivative and the parameters in the PCB.
PCData*	GetPCB()	Returns a pointer to the PCB. See table A.20 for a list of members.
void	Init()	Initializes the controller. Must be called before the first Control() call or if StopNorm, MaxIter, UWeight, UInit or YInit has been changed.
	PredictiveController(int ni, int no, int yorder, int uorder, int hs, int he, int hc, double uw)	The constructor for the class. ni is n_{in} the number of inputs. no is n_{out} the number of outputs. yorder is n the number of prev. outputs. uorder is m the number of prev. inputs. hs is h_s the Horizon Start parameter. he is h_e the Horizon End parameter. hc is h_c the Control Horizon parameter. uw is ρ the control signal cost weight.
void	SetTheModelUserData (void* UserData)	Sets the UserData pointer for the system model function and its derivative.

Table A.21: Predictive Controller Member Functions

A.8.4 NeuralPredictiveController Class Description

The class name is NeuralPredictiveController.

The NeuralPredictiveController class is a specialization of the PredictiveController class in which the system model is a neural network class object. It only adds two new member functions to specify and retrieve the network.

The Constructor has exactly the same parameters as the PredictiveController class constructor and can thus be seen in table A.21 in the previous section.

The two new methods that gets and sets a neural network pointer are listed and explained in table A.22.

Neural Predictive Controller Member Functions		
<R. Type>	Function	Description
NeuralNetwork*	GetTheNetwork()	Returns the system neural network pointer.
void	SetTheNetwork (NeuralNetwork* tn)	Sets the system model neural network pointer.

Table A.22: Neural Predictive Controller Member Functions

A.8.5 Neural Predictive Controller MATLAB MEX Function

A MATLAB MEX function has been developed in order to facilitate the simulation of a neural predictive controller on a SIMULINK model.

The call syntax of the MEX function is as follows.

```
NPC(System, T, W1, W2, YOrder, UOrder, HorizonStart,
    HorizonEnd, ControlHorizon, Rho, Reference,
    YInit, UInit, N, MaxIter
```

The parameters for the NPC MEX function are explained in table A.8.5.

System	The name of the SIMULINK system model.
T	The SIMULINK model sample time.
W1	The hidden layer weight matrix (See section A.7.1).
W2	The output layer weight matrix (See section A.7.1).
YOrder	The number of previous output signals utilized by the neural network system model.
UOrder	The number of previous input signals utilized by the neural network system model.
HorizonStart	The number of samples into the future from where the sum of squared errors begin.
HorizonEnd	The number of samples into the future where the sum of squared errors end.
ControlHorizon	The number of future control signals squared contributing to the cost function.
Rho	The weight of the future control signals squared in the cost function. (See equation A.20).
Reference	The reference signals chosen for the simulation.
YOrder	The initial values for the outputs (As in equation A.16).
UOrder	The initial values for the inputs (As in equation A.18).
N	The duration of the simulation in samples.
MaxIter	The maximum number of iterations for the optimizer used to find the optimal future control signals in the NPC.

Table A.23: NPC MEX Function Parameters

A.9 The Automatic Fallout Removal MATLAB Program

The following is a listing of the automatic fallout removal MATLAB function capable of utilizing a "support" signal to indicate where there naturally have to be a large change in the signal with fallouts. The function syntax is as follows.

```
y=removenoisespikes(x,wl1,nl1,z,wl2,nl2)
```

Where

`x`	Is the signal with fallouts in it.
`w11`	Is the number of samples around the fallout point which is utilized to calculate a mean value as the replacement value for the fallout.
`nl1`	Is the value the difference signal `xd` `xd=[0 x(2:end)-x(1:end-1)];`, should be larger than in order to consider the index a possible fallout index.
`z`	Is the "support" signal which must be some kind of input signal to the system where the signal x is coming from. It must indicate the natural points of excitation so that those index points will not be considered fallouts.
`w12`	Is the number of samples around each point in the "support" signal utilized to calculate the mean value of those `w12` points. Each point in the "support" signal is replaced by those mean values. This will smooth the support signal in order to make the spikes in the "support" signal difference signal wider. The wider the "support" signal difference signal spikes are the less points in x will be considered fallouts.
`nl2`	Is the value that the "support" signal difference signal should be larger than in order to make an index be considered a naturally occurring step.

The program listing follows, but the source code can also be found on the source code appendix CD coming with this dissertation in the folder ToolBoxes/NeuralSysId.

```
function y=removenoisespikes(x,wl1,nl1,z,wl2,nl2)

x=reshape(x,1,length(x));   % Convert input signals to
                            % row vectors.
z=reshape(z,1,length(x));
```

```
ex=x(2:end)-x(1:end-1);     % Compute the difference
                            % signal. Make it the
                            % original length by inserting
ex=[0 ex];                  % a zero at pos. 1.

if nl1<0                    % Calculate noise deviation
                            % limit unless it is already
                            % given as a positive number.
    nl1=-nl1*std(ex);
end

exi=find(abs(ex)>nl1);      % Find indices where the
                            % difference signal is larger
                            % than the noise deviation
                            % limit for the difference
                            % signal.

zs=msmooth(z,wl2);          % Smooth the indicating signal.

ez=zs(2:end)-zs(1:end-1);   % Calculate the difference
                            % signal for the smoothed
ez=[0 ez];                  % indicating signal. Make it
                            % the original length.
if nl2<0                    % Calculate the noise deviation
                            % limit for the smoothed
                            % difference indicating signal.
    nl2=-nl2*std(ez);
end

ezi=find(abs(ez)>nl2);      % Find the indices where the
                            % difference signal is larger
                            % than the noise deviation
                            % limit for the smoothed
                            % difference indicating signal.
different=setdiff(exi,ezi); % Find the indices where the
                            % signal difference is larger
                            % than the noise deviation
                            % limit for the difference
                            % signal but not those indices
                            % where the smoothed signal
                            % difference is larger than the
                            % noise deviation limit for the
                            % smoothed signal.
```

```
xs=msmooth(x,wl1);          % Smooth the signal.
x(different)=xs(different); % Replace fallouts with
                            % smoothed signal values.
y=x;
```

The `removenoisespikes` function utilizes another function called `msmooth`. It calculates a smoothed signal where each point in the input signal is replaced by the mean value of a specified number of points around the point. It has been written as a MATLAB MEX function and the source code can be found on the source code appendix CD in the folder ToolBoxes/Mex. The `msmooth` function has the following syntax.

```
y=msmooth(x,wl)
```

Where

- x Is the signal to be smoothed.
- wl Is the number of samples around each point utilized in the mean value calculation.

References

[1] Adams A. and Tsang C.P. Momentum in a back propagation artificial neural network. *AI '90. Proceedings of the 4th Australian Joint Conference on Artificial Intelligence*, pages 191–200, 1990.

[2] Alamir, M. and Bornard, G. Optimization based stabilizing strategy for non-linear discrete time systems with unmatched uncertain. *Second International Symposium on Methods and Models in Automation and Robotics*, 1995.

[3] Alamir, M. and Bornard, G. Stability of a Truncated Infinite Constrained Receding Horizon Scheme: the General Discrete Nonlinear Case. *Automatica, No. 9*, 31, 1995.

[4] Nikolaos Ampazis and Stavros J. Perantonis. Two highly efficient second order algorithms for training feed forward networks. *IEEE Transactions On Neural Networks*, 13(5):1064–1074, September 2002.

[5] Yixin Chen and Bogdan M. Wilamowski. A trust region based error aggregated training algorithm for neural networks. *IEEE*, pages 1463–1468, 2002.

[6] Xi Gang Zhang Jianwu Chen Li. On-Line State Prediction of Engines Based on Fast Neural Network. *SAE*, SP-1585:185–191, 2001.

[7] Cybenko, G. Approximation by Superpositions of a Sigmoidal Function. *Urbana University of Illinois*, 1988.

[8] Cybenko, G. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[9] Wendeker, M. Czarnigowski, J. Hybrid Air/Fuel Ratio Control Using the Adaptive Estimation and Neural Network. *SAE*, SP-1501:237–244, 2000.

[10] E. Hendricks, A. Chevalier, Michael Jensen, S. C. Sorenson, Dave Trumpy and Joe Asik. Modelling of the Intake Manifold Filling Dynamics. *SAETP*, -(960037), 1996.

[11] F. U. Dowla E. M. Johansson and D. M. Goodman. Backpropagation learning for multilayer feed-forward neural networks using the conjugate gradient method. *Int. J. Neural Syst.*, 2(4):38–51, 1991.

[12] M. Føns. Indførelse af Recirkulering af Udstødningsgas (EGR) i SI-Motorer. Master's thesis, The Technical University of Denmark, Department of Energy and Institute of Automation, DTU, 1997. In danish.

[13] Meng-Hock Fun and Martin T. Hagan. Levenberg-marquardt training for modular networks. *IEEE*, pages 468–473, 1996.

[14] Martin T. Hagan and Mohammad B. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions On Neural Networks*, 5(6):989–993, November 1994.

[15] Atkinson, C. M. Long, T.W. Hanzevack, E.L. Virtual Sensing: A Neural Network-based Intelligent Performance and Emissions Prediction System for On-Board Diagnostics and Engine Control. *SAE*, SP-1357:39–51, 1998.

[16] Haykin, S. *Neural Networks, A comprehensive Foundation*. Prentice Hall, 1994.

[17] E. Hendricks, T. Vesterholm, P. Kaidantzis, P. Rasmussen, and M. C. Jensen. Nonlinear Transient Fuel Film Compensation. *SAETP*, -(930767), 1993.

[18] R. A. Jacobs. Increase rate of convergences through learning rate adaption. *Neural Networks*, 1(4):295–308, 1988.

[19] Ole Tingleff Kaj Madsen. *Methods for Non-Linear Least Squares Problems*. IMM,DTU, November 1997.

[20] Mason, J.D. Kambhampati, C., Delgado, A. and Warwick,K. Stable receding horizon control based on recurrent networks. *IEE Proc.-Control Theory Appl., No. 3*, 144, 1997.

[21] Kazuyuki Ito Kiyotaka Shimizu. Design of Neural Stabilizing Controller for Nonlinear System via Lyapunov's Direct Method. *IJCNN*, CDROM, 1999.

[22] Zhijian James Wu Lee, A. Misfire Detection Using a Dynamic Neural Network with Output Feedback. *SAE*, SP-1357:33–37, 1998.

[23] G. Lera and M. Pinzolas. Neighborhood based levenberg-marquardt algorithm for neural network training. *IEEE Transactions on Neural Networks*, 13(5):1200–1203, September 2002.

[24] M. Føns, M. Müller, A. Chevalier, C. W. Vigild, S. C. Sorenson and E. Hendricks. Mean Value Engine Modeling of an SI engine with EGR. *SAETP*, -(1999-01-0909), 1999.

[25] Seungbum Park P. Yoon M Sunwoo. Feedback Error Learning Neural Networks for Spark Advance Control Using Cylinder Pressure. *Proc. Instn Mech Engrs*, 215, Part D, 2001.

[26] Grimaldi, C. N. Mariani, F. On Line Working Neural Estimator of SI Engines Operational Parameters. *SAE*, SP-1501:227–235, 2000.

[27] Marquardt, D. An Algorithm for least Squares Estimation of Non-Linear Parameters. *J.Soc. Ind. Appl. Math.*, 1963.

[28] Michalska, H. Mayne, D.Q. Receding Horizon Control of Nonlinear Systems. *IEEE Transactions on automatic control, No. 7*, 35, 1990.

[29] Michalska, H. Mayne, D.Q. Robust Receding Horizon Control of Constrained Nonlinear Systems. *IEEE Transactions on automatic control, No. 11*, 38, 1993.

[30] Ortmann, S. Rychetsky, M. Glesner, M. Groppo, R. Tubetti, P. Morra, G. Engine Knock Estimation Using Neural Networks Based on a Real-World Database. *SAE*, SP-1357:17–24, 1998.

[31] Nørgaard, P.M. *System Identification and Control with Neural Networks*. PhD thesis, DTU, 1996.

[32] Nørgaard, P.M. Neural network based system identification toolbox. Technical report, IAU, Technical University of Denmark, building 326, January 2000.

[33] Hellring, M. Munther, T. Rögnvaldsson, T. Wickström, N. Carlsson, C. Larsson, M. Nytomt, J. Robust AFR Estimation Using the Ion Current and Neural Networks. *SAE*, SP-1419:147–151, 1999.

[34] Hellring, M. Munther, T. Rögnvaldsson, T. Wickström, N. Carlsson, C. Larsson, M. Nytomt, J. Spark Advance Control Using the Ion Current and Neural Soft Sensors. *SAE*, SP-1419:153–159, 1999.

[35] Parisini, T. and Zoppoli, R. A Receding-horizon Regulator for Nonlinear Systems and a Neural Approximation. *Automatica, No. 10*, 31, 1995.

[36] Sørensen, P.H. Nørgaard, M. Ravn, O. Poulsen, N.K. Implementation of neural network based non-linear predictive control. *Internation Journal of Neurap*, 3, 1998.

[37] Gamo, S. O. Ouladsine, M. Rachidt', A. Diesel Engine Exhaust Emissions Modelling Using Artificial Neural Networks. *SAE*, SP-1419:161–166, 1999.

[38] Arsie, I. Marotta, F. Pianese, C. Rizzo, G. Information Based Selection of Neural Networks Training Data for S.I. Engine Mapping. *SAE*, SP-1585:173–184, 2001.

[39] Ronald Soeterboek. *Predictive Control, a unified approach*. Prentice Hall, 1992.

[40] Luther, J.B. Sørensen, P.H. Stability of a Neural Predictive Controller Scheme on a Neural Model. *IJCNN*, CDROM, 1999.

[41] Ayeb, M. Lichtenhäler, D. Winsel, T. Theuerkauf, H. J. SI Engine Modelling Using Neural Networks. *SAE*, SP-1357:107–115, 1998.

[42] Winsel, T. Ayeb, M. Lichtenhäler, D. Theuerkauf, H. J. A Neural Estimator for Cylinder Pressure and Engine Torque. *SAE*, SP-1419:179–194, 1999.

[43] Chih-Cheng Hung Venkata Atluri and Tommy L. Coleman. An artificial neural network for classifying and predicting soil moisture and temperature using levenberg-marquardt algorithm. *IEEE*, pages 10–13, 1999.

[44] C. W. Vigild, K. P. H. Andersen, E. Hendricks, and M. Struwe. Towards Robust H-infinity Control of an SI Engine's Air/Fuel Ratio. *SAETP*, -(1999-01-0854), 1999.

[45] S.F. McLoone V.S. Asirvadam and G.W. Irwin. Parallel and separable recursive levenberg-marquardt training algorithm. *IEEE*, pages 129–138, 2002.

[46] Lichtenhäler, D. Ayeb, M. Theuerkauf, H. J. Winsel, T. Improving Real-Time SI Engine Models by Integration of Neural Approximators. *SAE*, SP-1419:167–177, 1999.

[47] Parisini, T. Sanguineti, M. Zoppoli, R. Nonlinear stabilization by receding-horizon neural regulators. *Int. J. Control*, 3, 1998.